OpWeave: Flexible Operator Disaggregation for Heterogeneous LLM Serving

Zikun Li*
Carnegie Mellon University
zikunl@andrew.cmu.edu

Yixuan Mei*
Carnegie Mellon University
yixuanm@andrew.cmu.edu

Shiqi Pan
Carnegie Mellon University
shiqip@andrew.cmu.edu

Zixuan Chen
Carnegie Mellon University
zixuanc3@andrew.cmu.edu

Xiaowen Zhang
Carnegie Mellon University
xiaowen5@andrew.cmu.edu

Mengdi Wu
Carnegie Mellon University
mengdiwu@andrew.cmu.edu

Shuhuai Lin
Carnegie Mellon University
shuhuailinux@gmail.com

Yutong Yang
Carnegie Mellon University
aprilytyang@gmail.com

Zhihao Zhang
Carnegie Mellon University
zhihaoz3@andrew.cmu.edu

Xupeng Miao†
Peking University
xupeng.miao@pku.edu.cn

Rashmi Vinayak
Carnegie Mellon University and Google
rvinayak@andrew.cmu.edu

Zhihao Jia
Carnegie Mellon University
zhihao@cmu.edu

arXiv:2609.14237v1 [cs.DC] 13 Sep 2026

Abstract

LLM serving systems increasingly disaggregate inference into finer-grained stages, with recent approaches separating attention from FFN or MoE execution during decode. This *operator-level disaggregated serving* (ODS) can improve hardware matching and enable independent scaling, particularly across heterogeneous devices. However, existing systems fix operator boundaries and lack a unified characterization of when disaggregation reduces serving cost. We present OpWeave, an end-to-end framework for heterogeneous ODS. OpWeave provides an analytical cost model that bounds the gains of homogeneous and heterogeneous ODS over colocated serving. It jointly optimizes operator partitioning and deployment configuration through a regularity-aware planner that keeps the search tractable even for hybrid-attention models. A vLLM-based runtime executes the synthesized plans with flexible operator stages across heterogeneous device groups. In our evaluation, OpWeave reduces serving cost by up to 1.78× on homogeneous and 1.89× on heterogeneous GPU clusters relative to the best feasible baseline, while meeting latency SLOs.

1 Introduction

Large language model (LLM) inference accounts for a substantial share of AI infrastructure spending and requires significant investment in compute [36, 38, 52]. At this scale, even modest improvements in throughput and hardware utilization, or reductions in per-token cost, can yield substantial savings. Improving LLM serving efficiency has therefore become a central systems challenge [1, 23, 44, 54, 56].

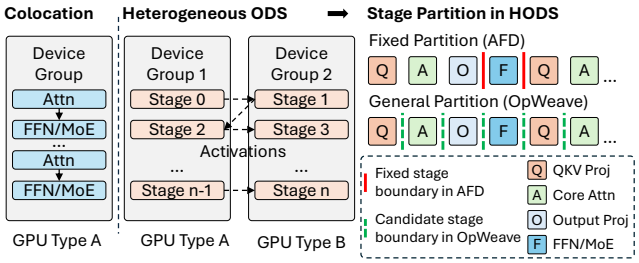


Figure 1. Colocated serving versus heterogeneous operator-level disaggregated serving (ODS). AFD fixes stage boundaries at attention–FFN interfaces, whereas OpWeave supports general operator partitions across heterogeneous device groups.

Modern serving systems increasingly separate inference phases to optimize and scale them independently. Most existing approaches disaggregate coarse-grained phases, particularly prefill and decode [10, 11, 33, 44, 48, 56]. However, these approaches generally retain decode as a single execution unit, despite substantial differences in the resource demands of its constituent operators. Recent systems extend disaggregation into the decode phase by separating attention from feed-forward network (FFN) or mixture-of-experts (MoE) execution [3, 31, 49, 50, 58]. This development motivates our focus on operator-level disaggregated serving.

Operator-level disaggregated serving (ODS) partitions decode computation into separately executed operator stages, such as attention and FFN or MoE stages. *Heterogeneous ODS* further assigns these stages to different device types, as illustrated in Figure 1. This design addresses two sources of inefficiency in conventional serving engines [1, 23, 54, 55].

*Zikun Li and Yixuan Mei contributed equally to this work.
†Work done at Purdue.

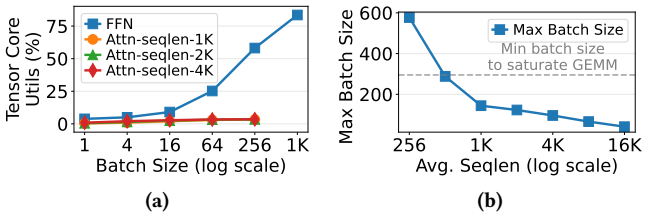


Figure 2. Decode-phase bottlenecks of Gemma-3-27B motivating ODS. *(a)* Tensor-core utilization of attention and FFN operators across batch sizes (B200): attention remains underutilized while batching improves FFN utilization. *(b)* Maximum feasible batch size across sequence lengths (H100): KV-cache growth constrains batching.

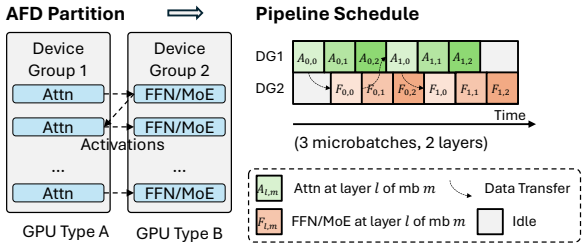


Figure 3. Pipelined attention–FFN disaggregation (AFD).

First, colocating operators can lead to *poor hardware matching*: decode attention is often limited by memory bandwidth and underutilizes tensor cores, while FFN and MoE operators rely heavily on matrix multiplications and benefit from high arithmetic throughput [50, 57, 58]. Second, colocation introduces *memory-capacity coupling*: as sequence length increases, the growing key–value (KV) cache limits the feasible batch size for the entire workload, potentially constraining FFN and MoE throughput.

Heterogeneous ODS mitigates these inefficiencies by pipelining operator stages across groups of heterogeneous GPUs [15, 31, 49, 50, 58]. First, it improves *hardware matching* by assigning each stage to devices suited to its resource requirements. Second, it enables *independent scaling*, allowing FFN and MoE stages to use batch sizes that are not directly constrained by the KV-cache capacity of an individual attention stage. Together, these capabilities can improve decode efficiency and reduce serving cost.

Despite this potential, realizing these benefits requires addressing three challenges.

Theory gap. Existing work provides empirical evidence and analytical models for specific ODS designs [31, 49, 50, 58]. A unified analysis is needed to determine when ODS reduces cost relative to colocated serving, establish bounds on the benefits of homogeneous and heterogeneous ODS, and characterize how these benefits depend on sequence length, model architecture, and hardware characteristics.

Planning gap. Existing systems optimize deployments around fixed operator boundaries, such as attention–FFN or attention–MoE splits [50, 58]. These fixed boundaries constrain joint optimization of operator partitioning, hardware assignment, parallelism, and batching. A general formulation is needed to synthesize efficient deployment plans across this broader design space, particularly for hybrid-attention models whose operator structures vary across layers.

Runtime gap. Existing ODS runtimes organize execution and communication around specific attention–FFN or attention–MoE splits [50, 58]. Executing synthesized plans with different operator boundaries and pipeline schedules requires flexible operator staging, efficient pipeline orchestration, and low-overhead communication across heterogeneous device groups.

To address these challenges, we present OpWeave, an end-to-end framework for heterogeneous ODS that integrates theoretical analysis, deployment planning, and runtime execution.

First, OpWeave uses an analytical cost model for colocated serving, homogeneous ODS, and heterogeneous ODS. The analysis characterizes how cost reductions depend on sequence length, model architecture, and hardware, and upper-bounds the gains of disaggregation and heterogeneous hardware assignment. Under this model, *core-attention disaggregation* (CAD) realizes the idealized ODS cost when its stages are balanced and communication is fully overlapped. The analysis also identifies two limitations of CAD: hardware and workload constraints can prevent stage balance, and layerwise variation in hybrid-attention models can introduce pipeline imbalance under a fixed two-stage partition.

Second, OpWeave formulates heterogeneous ODS planning as a joint optimization problem over operator partitioning, hardware assignment, parallelism, batching, and pipeline orchestration. To make this combinatorial search tractable, we introduce the *partition block*: the smallest repeating, layer-aligned operator pattern in a model, used as the unit of plan construction and reuse. Building on this abstraction, OpWeave develops a *regularity-aware* planner that reduces the search space by reusing partitioning decisions across layers with the same high-level operator structure, even when attention mechanisms differ. Together they support operator partitions beyond fixed attention–FFN or attention–MoE splits.

Third, OpWeave executes synthesized serving plans through a distributed runtime built on vLLM. The runtime's staged engines support flexible operator partitions and pipeline schedules across heterogeneous device groups. To reduce host orchestration overhead, each worker submits its computation and communication sequence in one native

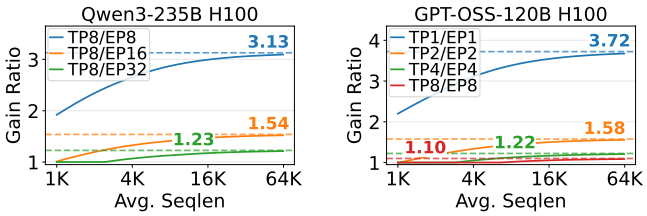


Figure 4. Theoretical cost gain of homogeneous ODS over colocated serving. Solid curves show gains for Qwen3-235B and GPT-OSS-120B on H100 under different attention-TP/MoE-EP configurations; dashed lines show upper bounds.

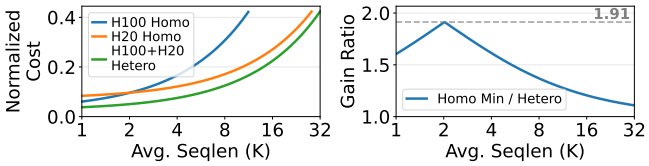


Figure 5. Heterogeneous vs. homogeneous ODS for Qwen3-235B on H100 and H20 GPUs. The heterogeneous setup runs core attention on H20 and GEMM-dominated operators on H100. *Left:* normalized cost per token. *Right:* cost gain over the best homogeneous deployment.

call per decoding iteration. Low-overhead communication further reduces transfer costs.

Finally, we evaluate OpWeave across representative models, hardware configurations, and long-context workloads. On homogeneous H100 clusters, OpWeave reduces serving cost by up to 1.78× compared to the best existing approaches while attaining time-per-output-token (TPOT) service-level objectives (SLOs). Compared with attention–FFN disaggregation (AFD), OpWeave plans require fewer pipeline stages (20.7 versus 124 on average for Gemma-3-27B) and reduce inter-node data transfer per output token by up to 22.8×; the resulting latency reduction lets OpWeave meet stringent TPOT SLOs that AFD cannot satisfy. OpWeave also keeps hardware executing the model over 80% of the time, whereas AFD leaves it idle roughly half the time. On heterogeneous clusters combining H100 with L40S or A100 GPUs, heterogeneous OpWeave reduces serving cost by up to 1.89× compared to the best non-OpWeave baseline and by up to 21.8% compared to homogeneous OpWeave.

2 Background and Motivation

Transformer Inference. Transformer-based LLMs stack repeated layers of attention and feed-forward computation, implemented as FFN or MoE blocks. Inference consists of two phases with distinct computational profiles: prefill processes the input context in parallel and is typically compute-bound, whereas decode generates one token per autoregressive iteration. This contrast motivates prefill-decode disaggregation, which places the two phases on separate resources for independent optimization [44, 56]. For generation-heavy workloads, such as reasoning tasks that produce long output sequences, decode often dominates latency and serving cost.

Decode-Phase Tensions. Serving systems batch concurrent requests to amortize model-weight accesses and improve hardware utilization, particularly for GEMM-heavy FFN and MoE operators. However, attention and FFN/MoE benefit differently from batching. Attention remains memory-bandwidth-bound and exhibits low tensor-core utilization even as batch size increases, as shown in Figure 2a. Moreover, as context length grows, the attention key-value (KV) cache reduces the maximum feasible batch size, as shown in Figure 2b, preventing FFN and MoE operators from reaching the batch sizes needed for efficient execution.

Attention–FFN Disaggregation. Prior heterogeneous ODS systems, including Step-3 and MegaScale-Infer, primarily adopt a fixed attention–FFN disaggregation (AFD) design rather than searching over general operator partitions [50, 58]. The attention stage executes the full attention module, including the QKV and output projections, while the FFN/MoE stage executes feed-forward or expert computation. Because attention and FFN/MoE alternate within each Transformer layer, hidden states must be transferred between the device groups at each stage boundary, as shown in Figure 3. These systems divide batches into microbatches and pipeline their execution to keep both stages busy while overlapping inter-stage communication with computation. Consequently, performance depends strongly on stage balance: latency mismatches create pipeline bubbles and reduce utilization. Although prior systems demonstrate the feasibility of AFD, their fixed partitions leave unresolved when heterogeneous ODS reduces serving cost and what determines its benefit, the question we address next through theoretical analysis.

3 Theoretical Analysis

3.1 Analytical Setup

Scope and assumptions. We analyze steady-state decode serving with memory-bound attention and GEMM-dominated projections, FFN, and MoE computation. Each GPU type has unit-time monetary cost c, memory bandwidth β, memory capacity M, and peak compute throughput F; we minimize GPU cost per decoded token. We model attention execution time as memory traffic divided by memory bandwidth, and GEMM execution time as FLOP count divided by peak GPU compute throughput only in the compute-bound regime. We assume disaggregation enables GEMM batches large enough for compute-bound execution, and that microbatching and pipelining hide inter-device communication. Appendices A.1 and A.2 give the operator-level roofline model and formal regime assumptions.

Cost model. Let $m_{\mathrm{kv}}(s)$ be the KV-cache memory per request at average sequence length s. Let P_{act} be the effective number of GEMM-side parameters activated per request after model-parallel sharding, and let M_{weights} be the resident model-weight footprint on the modeled device group, including inactive MoE experts. For GPU type j, define $\alpha_j \triangleq c_j/\beta_j$ and $\gamma_j \triangleq c_j/F_j$ as monetary cost per byte of memory traffic and per FLOP, respectively. Under the regime assumptions above, homogeneous ODS on type j has cost

$$\mathrm{CPT}_j^{\mathrm{hom}}(s) = \alpha_j m_{\mathrm{kv}}(s) + 2\gamma_j P_{\mathrm{act}}, \tag{1}$$

while heterogeneous ODS with attention on GPU type j and GEMM-dominated operators on type k has cost

$$\mathrm{CPT}_{j,k}^{\mathrm{het}}(s) = \alpha_j m_{\mathrm{kv}}(s) + 2\gamma_k P_{\mathrm{act}}. \tag{2}$$

For colocated serving on type j with memory capacity M, define the continuous relaxation $\bar{b}(s)$ and the exact integer batch limit $b_{\max}(s)$ as

$$\bar{b}(s) \triangleq \frac{M - M_{\mathrm{weights}}}{m_{\mathrm{kv}}(s)}, \qquad b_{\max}(s) = \lfloor \bar{b}(s) \rfloor. \tag{3}$$

For feasible sequence lengths, the minimum colocated cost under the continuous batch-size relaxation is

$$\mathrm{CPT}_j^{\mathrm{coloc}}(s) = \alpha_j m_{\mathrm{kv}}(s) + \max\left(\frac{\alpha_j M_{\mathrm{weights}}}{\bar{b}(s)},\ 2\gamma_j P_{\mathrm{act}}\right). \tag{4}$$

The largest relaxed batch $\bar{b}(s)$ attains this minimum because cost is non-increasing in batch size. The maximum captures whether GEMM is limited by weight loading or compute.

GEMM threshold. For local batch size b, Eqs. 1–4 use the approximations that attention traffic scales as $b\, m_{\mathrm{kv}}(s)$, active GEMM work as $2bP_{\mathrm{act}}$, and GEMM weight traffic as M_{weights}. Equating GEMM weight-loading time and compute time gives the threshold $b^* = \frac{M_{\mathrm{weights}} F}{2P_{\mathrm{act}}\beta}$, which is the batch size at which GEMM becomes compute-bound. Appendix A.3 derives the cost expressions, and Appendix A.4 states the scaling approximations and threshold derivation in full.

3.2 Homogeneous Disaggregation vs. Colocation

We first analyze when homogeneous operator-level disaggregation reduces cost relative to colocation, and how that gain depends on sequence length. We fix the GPU type and omit its index. We restrict the comparison to sequence lengths for which colocation is feasible, i.e., $b_{\max}(s) \geq 1$, equivalently $\bar{b}(s) \geq 1$, and assume that $m_{\mathrm{kv}}(s)$ is strictly increasing on this domain. Define the *cost gain* of homogeneous disaggregation over colocation as $G_{\mathrm{hom}}(s) \triangleq \frac{\mathrm{CPT}^{\mathrm{coloc}}(s)}{\mathrm{CPT}^{\mathrm{hom}}(s)}$.

Theorem 3.1 (Cost gain of homogeneous disaggregation over colocation)**.** *Consider a homogeneous GPU setting with parameters (c, β, M, F). Suppose attention is memory-bound and adopt the scaling approximations from Section 3.1. Let b^* be the GEMM compute-bound threshold (Section 3.1) and let $\bar{b}(s)$ be the relaxed feasible batch size defined in Eq. 3. Then, for every feasible sequence length s with $b_{\max}(s) \geq 1$:*

1. (No gain at short sequences.) *If $\bar{b}(s) \geq b^*$, then $G_{\mathrm{hom}}(s) = 1$.*
2. (Strictly increasing gain.) *On the feasible region where $\bar{b}(s) < b^*$, $G_{\mathrm{hom}}(s) > 1$ and G_{hom} is strictly increasing in s.*
3. (Bounded gain.) *The gain satisfies*

$$1 \leq G_{\mathrm{hom}}(s) < \frac{M}{M - M_{\mathrm{weights}}}. \tag{5}$$

Homogeneous disaggregation helps only after KV-cache growth pushes the colocated feasible batch size below the GEMM compute-bound threshold. Beyond that point, the gain increases monotonically with sequence length but remains bounded by $M/(M - M_{\mathrm{weights}})$. Disaggregation improves efficiency by decoupling GEMM batching from KV-cache memory pressure, so its benefit is fundamentally limited by how much device memory is already consumed by model weights. Figure 4 shows cost gains and upper bounds calculated from our theoretical model for Qwen3-235B and GPT-OSS-120B on H100 under different parallelism configurations. See Appendix A.5.1 for the proof.

3.3 Heterogeneous vs. Homogeneous Disaggregation

Optimizing Eqs. 1–2 over all available GPU types j and k gives

$$\mathrm{CPT}_\star^{\mathrm{hom}}(s) = \min_j \mathrm{CPT}_j^{\mathrm{hom}}(s), \tag{6}$$

$$\mathrm{CPT}_\star^{\mathrm{het}}(s) = \min_j(\alpha_j)\, m_{\mathrm{kv}}(s) + \min_k(\gamma_k)\, 2P_{\mathrm{act}}. \tag{7}$$

The optimized heterogeneous deployment assigns attention to the GPU type with the lowest cost per byte and GEMM-dominated operators to the type with the lowest cost per FLOP; the homogeneous deployment uses one type for both.

Define $G_{\mathrm{het}}(s) \triangleq \frac{\mathrm{CPT}_\star^{\mathrm{hom}}(s)}{\mathrm{CPT}_\star^{\mathrm{het}}(s)}$ as the gain over homogeneous disaggregation. For the theorem below, restrict these minima to $j, k \in \{1, 2\}$ and define $\eta \triangleq (\beta_1 F_2)/(\beta_2 F_1)$ as the hardware ratio. Assume $\eta \geq 1$ without loss of generality. Let $\mathcal{I}$ be a sequence-length interval where all compared ODS deployments are feasible and $m_{\mathrm{kv}}(s)$ is continuous and strictly increasing.

Theorem 3.2 (Cost gain of heterogeneous over homogeneous disaggregation). *Consider two GPU types with parameters (β_1, F_1) and (β_2, F_2), and respective unit-time costs c_1 and c_2. Adopt the scaling approximations of Section 3.1. Then, over the feasible interval $\mathcal{I}$:*

1. (Non-negative gain.) *For every $s \in \mathcal{I}$, $G_{\text{het}}(s) \geq 1$.*
2. (No gain under dominance.) *If one GPU type dominates the other on both cost-efficiency metrics, i.e., $\alpha_j \leq \alpha_k$ and $\gamma_j \leq \gamma_k$ for some $j \neq k$, then $G_{\text{het}}(s) = 1$ for every $s \in \mathcal{I}$.*
3. (Unimodal and bounded gain.) *Otherwise, under the labeling $\eta \geq 1$, we have $\alpha_1 < \alpha_2$ and $\gamma_2 < \gamma_1$. Define the homogeneous-cost crossing*

$$x^* \triangleq \frac{2P_{\text{act}}(\gamma_1 - \gamma_2)}{\alpha_2 - \alpha_1}.$$

As a function of $x = m_{\text{kv}}(s)$, the gain strictly increases for $x < x^*$ and strictly decreases for $x > x^*$. If $x^* \in m_{\text{kv}}(\mathcal{I})$, the gain therefore has a unique peak at the corresponding $s^* \in \mathcal{I}$; otherwise, it is monotone over $\mathcal{I}$. Moreover, for every $s \in \mathcal{I}$,

$$G_{\text{het}}(s) \leq \frac{1 + \sqrt{\eta}}{2}. \tag{8}$$

The theorem shows that heterogeneous assignment helps only when the two GPU types trade off bandwidth efficiency against compute efficiency, rather than one dominating on both dimensions. When the homogeneous-cost crossing lies in the feasible interval, heterogeneity helps most there: at shorter sequences the best homogeneous GPU already matches the compute side well, while at longer sequences the best homogeneous GPU already matches the bandwidth side well. If the crossing lies outside the feasible interval, only the increasing or decreasing branch appears in that interval. Figure 5 shows normalized costs per token and the gain of heterogeneous disaggregation over the best homogeneous deployment, calculated from our theoretical model for Qwen3-235B using H100 and H20 GPUs. The full proof is given in Appendix A.5.2.

3.4 From Bounds to Plans: CAD and Its Limits

The bounds above assume that attention and the remaining operators run on hardware suited to their bottlenecks and that communication is fully hidden by computation. Core-Attention Disaggregation (CAD) implements this separation by placing core attention on one device group and the remaining projection, FFN, and MoE GEMMs on another. Under the assumptions in Section 3.1, attaining the corresponding idealized ODS cost requires balanced stage latencies and sufficient microbatching to hide communication (Appendix A.3).

The first limitation, however, is whether stage balance is feasible. GPU memory must accommodate the KV caches of all resident microbatches and attention layers, constraining each attention operation's KV-cache size and hence its maximum latency under our memory-bound model. GEMM-based operators, meanwhile, must load their weights from HBM, imposing a latency floor that our compute-bound cost model does not capture. If this floor exceeds the maximum feasible attention latency, the two stages' feasible latency ranges do not overlap, and the attention stage remains idle for part of every pipeline cycle. Higher memory bandwidth on the GEMM devices lowers this floor and can therefore make stage balance feasible. Figure 6 illustrates both cases, and Appendix A.6 derives the overlap condition for uniform-layer models.

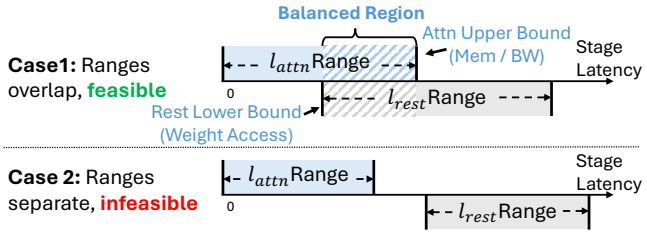


Figure 6. Feasibility of balanced CAD. *Case 1:* the attention and rest-of-model latency ranges overlap, admitting a balanced operating point. *Case 2:* the ranges are disjoint, so pipeline bubbles are unavoidable.

The NVIDIA Vera Rubin and Groq 3 LPX design illustrates this principle: it retains the KV cache and decode attention on GPUs while placing FFN or MoE weights in LPX's SRAM [3]. SRAM residency reduces weight-loading latency, lowering the FFN/MoE stage's latency floor and potentially enabling its feasible latency range to overlap with the attention stage's. Although this design uses AFD rather than CAD, the same two-stage balance principle applies.

The second limitation is layerwise variation. Hybrid architectures commonly mix full attention with sliding-window, linear, or recurrent sequence operators [2, 12, 22, 43, 47]. Full attention loads KV cache for the entire context, whereas efficient operators access a bounded window or compact recurrent state. CAD places both on the same attention device group, but the rest-of-model stage has one fixed latency for a given microbatch size and cannot match both layer types. This mismatch leaves one stage waiting for the other at some layers, creating the pipeline bubbles shown in Figure 7.

4 Serving-Plan Formulation

CAD's limitations motivate searching over general operator partitions rather than fixed module splits. Our formulation jointly determines how operators are partitioned into stages, which device-group types execute them with what hardware and parallelism, and how resident batches and microbatches are sized. It evaluates each plan using a steady-state latency model that accounts for both stage execution and inter-stage activation transfers, and minimizes serving cost subject to memory and TPOT constraints.

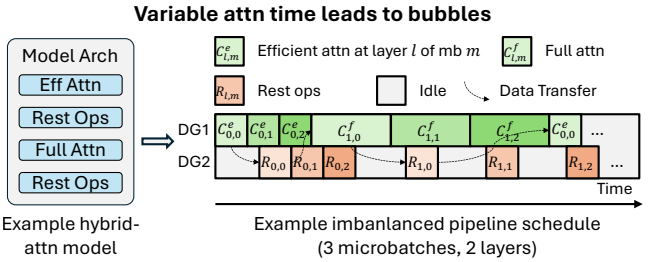


Figure 7. CAD pipeline schedule for a hybrid-attention model. A fixed rest-of-model stage cannot balance both efficient- and full-attention layers, leaving idle pipeline slots.

Partition-block abstraction. The key challenge is to capture cross-layer heterogeneity without searching an unconstrained partition of the entire model. We address it with a partition block, the smallest layer-aligned operator pattern that repeats across the model. A partition block may contain one layer in a uniform transformer or span a full period of full-attention, sliding-window, and other layer types in a hybrid-attention model. Within a block of N_{block} ordered operators, the planner chooses a contiguous stage template $\mathbf{q} = (q_0, \ldots, q_S)$, where $0 = q_0 < \cdots < q_S = N_{\text{block}}$ and stage j contains operators $q_{j-1}+1, \ldots, q_j$. Stages may span layer boundaries, allowing the planner to balance heterogeneous operators jointly across layers. Reusing the resulting template across all partition blocks preserves the model's regularity and reduces the partition search to one repeating unit. Each stage then serves as the basic unit of placement and pipeline execution; Appendix B.1 provides the formal DAG, ordering, and block definitions.

Structured placement and batching. Given the stage template, the planner determines where each stage executes and with what resources. We represent the deployment using K ordered device-group types. Type m has configuration (h_m, π_m), consisting of a GPU type and a supported parallelism strategy such as tensor or expert parallelism, and may be instantiated by multiple **replicas**. All replicas of a type execute the same assigned stage positions while processing disjoint subsets of the global batch. To obtain a reusable pipeline structure, we require $S = kK$ for some positive integer k and assign stage j using the structured round-robin map $\sigma(j) = ((j-1) \bmod K) + 1$. Each type therefore owns the same k stage positions in every partition block, so one template and pipeline schedule serve the entire model even though the stages' operator contents and execution costs may differ. A replica of type m has resident batch size b_m, divided into μ microbatches of size $\widehat{b}_m = b_m/\mu$.

Communication-aware latency model. Plan latency depends on both stage execution and communication across stage boundaries. For each stage j, let $t_j = f_{\text{lat}}(\cdot)$ denote its profiled execution latency under the assigned device-group configuration and microbatch size. Let $u_j = f_{\text{comm}}(\cdot)$ denote the profiled transfer latency after stage j, including any redistribution required by the source and destination parallel layouts. The boundary after stage S connects to stage 1 of the next partition block and is omitted after the final block. With B_{total} repeated partition blocks, the steady-state pipeline round-trip time is $T_{\text{rtt}} = f_{\text{rtt}}(\mu, B_{\text{total}}, \mathbf{t}, \mathbf{u})$, where $\mathbf{t} = (t_1, \ldots, t_S)$ and $\mathbf{u} = (u_1, \ldots, u_S)$. We evaluate f_{rtt} by schedule simulation, which enforces execution and transfer dependencies, models resource contention, and permits computation–communication overlap only when the schedule and resources allow. Appendices B.2 and B.3 provide the complete placement, batching, execution, and communication definitions.

Design-space expressiveness. The planner can isolate expensive full-attention operators, combine efficient-attention operators with neighboring projections, or map repeated stage positions to heterogeneous device-group types while reusing one block-level template. This flexibility directly addresses the layerwise imbalance of Section 3.4. AFD and CAD are recovered as restricted two-way templates: AFD separates complete attention modules from FFN or MoE operators, whereas CAD isolates only the core attention kernels; colocated serving is the single-stage template.

Optimization problem. A serving plan is specified by

$$\mathcal{P} = \left(S, K, \mathbf{q}, \left((h_m, \pi_m, b_m)\right)_{m=1}^{K}, \mu\right),$$

where $\mathbf{q}$ is the stage template defined above. The variables S, K, μ, and b_m are positive integers with $K \mid S$ and $\mu \mid b_m$ for every type m. Each h_m is drawn from the available hardware types, and π_m is a parallelism strategy supported by h_m. The indexed sequence of device-group configurations is ordered because stage j is assigned to type $\sigma(j)$.

$$\min_{\mathcal{P}} \quad \text{CPT}_{\text{plan}} = T_{\text{rtt}} \sum_{m=1}^{K} \frac{c_m}{b_m} \tag{9a}$$

$$\text{s.t.} \quad T_{\text{rtt}} \le T_{\text{SLO}}, \tag{9b}$$

$$M_{\text{weights},m} + \mu\, \text{KV}_m(\widehat{b}_m, s) \le M_m, \qquad m = 1, \ldots, K. \tag{9c}$$

The objective in equation 9a minimizes serving cost per generated token. For device-group type m, c_m is the unit-time cost of one replica. Thus, $\sum_m c_m/b_m$ is the aggregate cost rate per global resident request; since each resident request produces one token per pipeline round trip, multiplying by T_{rtt} gives the steady-state cost per token. Appendix B.5 derives this objective from the global batch size and replica counts. The constraints require the pipeline round-trip time to meet the TPOT SLO and each replica to fit its assigned model weights and the resident KV cache for all μ microbatches at sequence length s; Appendix B.4 provides the complete memory notation.

Coupled search challenge. The planning decisions cannot be optimized independently: stage boundaries, device-group configurations, parallelism strategies, resident batch sizes, and microbatch count jointly determine execution latency, transfer overhead, memory feasibility, and replica count. A lower-cost device may increase total cost if its smaller feasible batch size requires more replicas, while a compute-balanced partition may miss the TPOT SLO after boundary transfers are scheduled. Section 5 exploits partition-block regularity and Pareto structure to search this coupled space efficiently.

5 Algorithms

The serving-plan search in Section 4 is difficult for three reasons. First, independently placing every legal stage boundary creates a combinatorial partition space. Second, each device-group type, which we call an *owner* because it owns fixed stage positions in every partition block, admits many hardware, parallelism, and batching configurations. Third, the cross-owner product of these configurations is too large to enumerate and simulate exhaustively. Our planner addresses them in order: it parameterizes partitions with regularity-aware sub-block templates, constructs a network-safe Pareto frontier for each owner, and combines the surviving frontiers using exact branch-and-bound. Only combinations that survive all three reductions reach the schedule simulator.

5.1 Regularity-Aware Sub-Block Templates

An unrestricted partition that chooses every stage boundary independently produces $\binom{N_{\text{block}}-1}{S-1}$ templates for a block with N_{block} operators and S stages, even though the same operator order often repeats across many layers. This redundancy is especially large in hybrid-attention models: layers may use different attention variants while retaining the same sequence of attention projections, attention core, and FFN or MoE operators. We remove it by searching a smaller *sub-partition block*. Let L_{block} be the number of layers in a partition block, and let L_{sub} be a sub-block length in layers that divides L_{block} and preserves the repeated layer pattern; each block then contains $n = L_{\text{block}}/L_{\text{sub}}$ sub-block repetitions. For a chosen number of owners K, the planner selects a legal contiguous template $\mathbf{q}_{\text{sub}} = (q_0, \ldots, q_K)$ inside one sub-block and tiles it over the n repetitions. Stage position m in every repetition is assigned to owner m, giving $S = nK$ stages per partition block (Section 4's $S = kK$ with $k = n$). This parameterization reduces the number of templates to $\binom{N_{\text{block}}/n-1}{K-1}$ while retaining layer-dependent execution costs after the template is expanded; it is a structural restriction of the searched plan family, not a cost-based pruning heuristic.

5.2 Network-Safe Per-Owner Pareto Frontiers

The structural reduction still leaves many configurations for every owner. Given a template, global resident batch B, and microbatch count μ, a feasible point p for owner m chooses a hardware and parallelism configuration and a per-replica microbatch size $\widehat{b}_m(p)$, which fix an integral replica count. Profiling gives the execution latencies $\mathbf{T}_m(p) = \big(t_j(p)\big)_{j \in \mathcal{S}_m}$ of the stage positions $\mathcal{S}_m = \{j : \sigma(j) = m\}$ owned by m. The point also has normalized cost and GPU coordinates $r_m(p) = c_m(p)/b_m(p)$ and $\rho_m(p) = g_m(p)/b_m(p)$, where $b_m(p) = \mu\widehat{b}_m(p)$ and $g_m(p)$ is the GPU count per replica.

Latency and cost alone are insufficient for safe local pruning. Two points with similar execution latency can induce different replica counts, total GPU use, and per-endpoint transfer sizes when combined with neighboring owners. We therefore compare points using

$$\mathbf{z}_m(p) = \big(\mathbf{T}_m(p), r_m(p), \rho_m(p), \widehat{b}_m(p)\big). \tag{10}$$

Point p dominates point p' only if both use the same tensor-parallel degree, p is no worse in every coordinate of equation 10, and it is strictly better in at least one; the degree

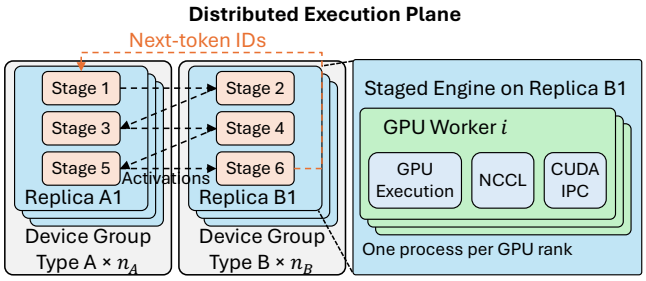


Figure 8. Distributed execution plane. Black dashed arrows carry activations between stages; the orange dashed path returns next-token IDs to the first-stage workers.

Algorithm 1 Regularity-Aware Exact Serving-Plan Search

Require: Model operator pattern, finite search grid, profiles, TPOT SLO
Ensure: Cheapest feasible plan in the structured space
best $\leftarrow \infty$
for each regular sub-block and legal template $\mathbf{q}_{\text{sub}}$ **do**
  Tile $\mathbf{q}_{\text{sub}}$ over the model
  for each feasible (B, μ) **do**
    Build network-safe frontiers $\mathcal{F}_1, \ldots, \mathcal{F}_K$ from memory-feasible owner points
    Explore$(1, \emptyset)$
return best plan
procedure Explore(i, P)
  if optimistic resource and cost bounds for P rule out a feasible improvement over best **then return**
  if $i > K$ **then**
    Tighten bounds; if P survives, simulate and update best if feasible and improved
    return
  for each $p \in \mathcal{F}_i$ **do**
    Explore$(i + 1, P \cup \{p\})$

restriction is needed because transfer volume varies non-monotonically with tensor parallelism. The GPU coordinate preserves the planner's preference for fewer GPUs among otherwise equal plans; the microbatch coordinate preserves the inputs of the batch-dependent transfer model. This dominance relation yields the network-safe frontier $\mathcal{F}_m$. Retaining $\mathbf{T}_m$ as a per-occurrence vector prevents a point that accelerates one layer type but slows another from being removed, so the frontier eliminates only configurations that cannot improve any feasible complete plan.

5.3 Exact Branch-and-Bound over Frontier Products

Explicitly evaluating $\mathcal{F}_1 \times \cdots \times \mathcal{F}_K$ can still require exponentially many schedule simulations, so the planner explores the product with depth-first branch-and-bound. For each prefix of assigned owners, it forms an optimistic completion from the best remaining coordinates of the unassigned frontiers: mandatory serialized compute work lower-bounds each owner's latency contribution, and mandatory transfer work at each network endpoint lower-bounds communication, with unassigned endpoints minimized independently over feasible replica counts, keeping the estimate optimistic. The larger of the two is a valid lower bound on T_{rtt}, and multiplying it by the suffix-minimal normalized cost lower-bounds the objective. A subtree is discarded if its optimistic completion violates the TPOT SLO or cannot improve the incumbent. Once every owner is assigned, the bounds are recomputed with the complete configuration, and only survivors invoke the schedule simulator. Appendix C.1 states these bounds formally.

The schedule simulator remains authoritative for every surviving candidate, so the planner is exact over the configured finite hardware, batching, and template grid; its worst-case complexity remains $\prod_m |\mathcal{F}_m|$ when no bound prunes.

6 System Design and Implementation

OpWeave is implemented in 32K lines of Python and C++ on the vLLM GPU model runner [23]; each worker loads only its assigned model components and state. Inter-node activation transfers use point-to-point NCCL, and intra-node fanout uses CUDA interprocess communication (IPC) peer copies with GPU doorbells. The planner-selected batch size fixes the plan's scheduling capacity: logical slots, microbatch shapes, stage placement, and per-rank action order are static, so CUDA graphs and buffers are bound once, while token, position, and KV-cache metadata change between iterations.

6.1 System Overview and Execution Flow

Runtime organization. Figure 8 shows the distributed execution plane after plan installation. A OpWeave deployment consists of a global coordinator (outside the figure) and replicas of the device-group types selected by the plan; each replica runs a staged engine with one GPU worker process per GPU rank. The coordinator acts only at setup, instantiating the replicas and installing the plan; during decoding, workers execute the compiled stage schedule without coordinator or RPC intervention between stages and microbatches.

Table 1. GPU specifications [39–41]. Prices are averaged across selected public clouds.

GPU type	Memory BW (GB/s)	Memory (GB)	BF16 (TFLOPS)	Price ($/GPU-h)
H100 SXM	3,350	80	989	3.49
L40S	864	48	362.05	1.09
A100 SXM	2,039	80	312	1.74

Request and stage execution. First-stage workers consume the current token IDs of batched requests. Each microbatch traverses the ordered stages, transferring activations whenever consecutive stages reside on different replicas; terminal workers produce next-token IDs, which return to the first-stage workers on the GPU (Section 6.2). A stage runs once its input activation has arrived, enforced by receive-completion events, and the preceding compute action in its worker's fixed action order has completed; independent computation and communication still overlap.

6.2 Optimizations

Fine-grained disaggregation creates many short compute and transfer actions per token. If each traverses Python or becomes a separate network operation, its overhead can erase the benefit of finer placement. We apply three optimizations.

Single native submission. A naive executor returns to Python for every stage, send, and receive, placing host dispatch between otherwise short GPU actions. Instead, OpWeave compiles the complete per-rank sequence; for each token, one native submission enqueues all CUDA-graph launches, staging copies, communication rounds, and event dependencies, eliminating Python dispatch, allocation, and RPCs between stages and microbatches.

GPU-resident decoding. A centralized implementation would copy generated token IDs to the host, assemble and redistribute them, and rebuild execution metadata at every step, serializing a CPU–GPU round trip into each token. In OpWeave, terminal workers compute token IDs on the GPU, one NCCL all-reduce distributes the token vector to embedding owners, and tokens stay device-resident between iterations; only global rank zero retains the token history for eventual host-side output. Each stage reuses preconstructed metadata and preallocated buffers, updating only the tensors that change between iterations.

Low-overhead communication. Our runtime packs compatible tensors into shared buffers and batches transfers into

Table 2. Evaluated models and TPOT SLOs. Eff. Attn. is the efficient attention mechanism interleaved with full attention; Pattern is the ratio of efficient- to full-attention layers.

Model	MoE	Eff. Attn.	Pattern	Strict SLO (ms)	Relaxed SLO (ms)
Gemma-3-27B	✗	SWA	5:1	25	60
Qwen3-Next-80B-A3B	✓	Linear	3:1	40	80

NCCL rounds, reducing per-operation overhead without increasing activation payload. When an unsharded activation crosses a stage boundary into a TP group, the runtime sends one inter-node copy to a destination leader rather than one copy per rank; the leader fans it out to the remaining TP ranks locally through CUDA IPC, and leader assignment rotates across routes to balance cross-node traffic.

7 Evaluation

7.1 Evaluation Setup

Devices and Models. Table 1 lists the GPUs used in our evaluation. We conduct the homogeneous evaluation on a cluster of H100 nodes connected by RDMA over Converged Ethernet (RoCE). For the simulator fidelity tests, we use H100 and L40S GPUs on AWS, where RDMA transfers stage GPU data through host memory. We evaluate two representative hybrid-attention models [12, 47], with their architectures and TPOT SLOs summarized in Table 2.

Metrics. We use serving cost per million output tokens as our primary efficiency metric and time per output token (TPOT) as our latency metric. A configuration satisfies a target TPOT service-level objective (SLO) if its p95 step-level TPOT across the measured iterations does not exceed the target; for each serving policy, we report the lowest-cost configuration that satisfies the SLO.

Baselines. We compare OpWeave against three baselines: colocated serving (COL), attention–FFN disaggregation (AFD) [50, 58], and core-attention disaggregation (CAD). COL is implemented with vLLM 0.26.0 [23] and executes all operators on GPUs within the same node. AFD and CAD are implemented in our OpWeave runtime as fixed-partition special cases. AFD separates complete attention modules from FFN/MoE operators, whereas CAD isolates core-attention kernels from all remaining operators; their two device groups may be placed on different nodes. This setup gives COL a well-optimized production implementation and keeps the runtime identical across disaggregated policies.

Search Configuration. For each model, workload, and TPOT SLO, our planner searches the serving-plan space of Section 5 for the lowest-cost plan meeting the SLO. We restrict each plan to at most 32 GPUs, three device-group types, four replicas per type, and four microbatches. OpWeave and all baselines use the same search configuration and resource constraints; their admissible plans differ only in the imposed disaggregation policy. The planner returns the optimal feasible plan within this configured finite search space.

Workloads. We evaluate steady-state decoding using long-context workloads derived from OpenThoughts3-1.2M [14]. We tokenize the dataset to obtain an empirical distribution of generation-time context lengths and rescale it to construct synthetic request cohorts with mean context lengths of 8K, 32K, and 128K tokens. Requests are constructed deterministically, and all serving policies use the same distribution and construction procedure, ensuring comparable workloads.

7.2 Homogeneous End-to-End Evaluation

We evaluate OpWeave and all baselines on H100 GPUs across both models, both SLOs (Table 2), and all three context lengths. As shown in Figure 9, OpWeave achieves the lowest serving cost in 11 of the 12 evaluated model–context–SLO settings. OpWeave is up to 1.78× cheaper than the best feasible baseline for Gemma-3-27B and up to 1.76× cheaper for Qwen3-Next-80B-A3B.

OpWeave finds feasible plans in all evaluated scenarios and achieves lower serving costs than AFD and CAD wherever they are feasible. *Its first advantage is lower latency through fewer pipeline stages.* Whereas AFD and CAD impose two stages per layer, OpWeave selects stage boundaries across an entire partition block, enabling a larger, more flexible partition space. As shown in Figure 10(a) and (d), the selected OpWeave plans average only 20.7 stages for Gemma-3-27B and 24 for Qwen3-Next-80B-A3B, compared with 124 and 96, respectively, for both baselines. Fewer stage boundaries reduce network transfers: Figure 10(b) and (e) show that OpWeave reduces mean inter-node data transferred per output token by factors of 16.9 and 22.8 relative to CAD and AFD, respectively, for Gemma, and by factors of 6.5 and 6.1 for Qwen3-Next. Because inter-stage communication lies on the execution critical path of each microbatch, this reduction lowers decoding latency, helping OpWeave satisfy strict TPOT SLOs in settings where neither baseline finds a feasible plan.

OpWeave's second advantage is higher hardware occupancy by model execution, reducing idle time during decoding. We report system-level occupancy as the average of per-replica occupancy, weighting each replica type by its price times its replica count, i.e., the fraction of hardware spending that performs model execution. As shown in Figure 10(c) and (f), OpWeave achieves mean compute occupancy above 80% for both models, while CAD and AFD leave roughly half of the hardware time unused. One reason for the baselines' lower occupancy is heterogeneity across layers: the selected hybrid models interleave attention mechanisms with substantially different execution latencies, making their fixed partitions

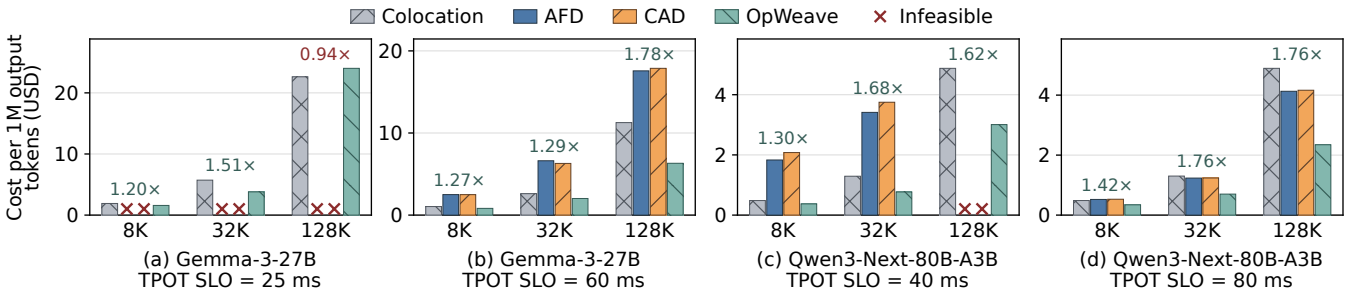


Figure 9. Measured serving cost across context lengths and TPOT SLOs. Crosses mark infeasible configurations. Labels above each group show $g\times$ cheaper, where g is the best feasible baseline cost divided by OpWeave cost.

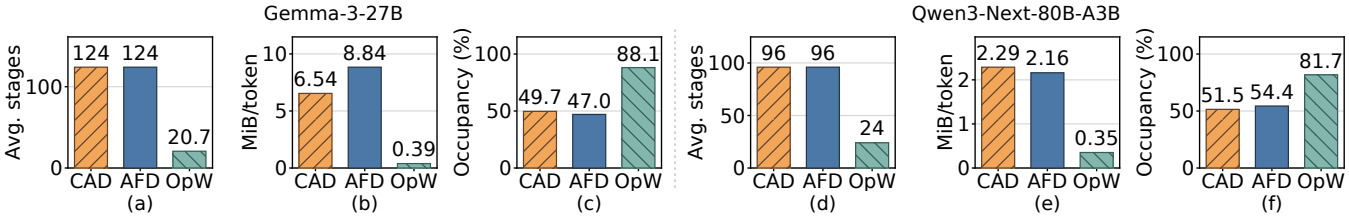


Figure 10. Homogeneous-machine evaluation results. OpW denotes OpWeave. (a) and (d): Mean number of pipeline stages. (b) and (e): Mean inter-node stage-to-stage payload per output token (MiB). (c) and (f): Mean price-weighted compute occupancy (Section 7.2).

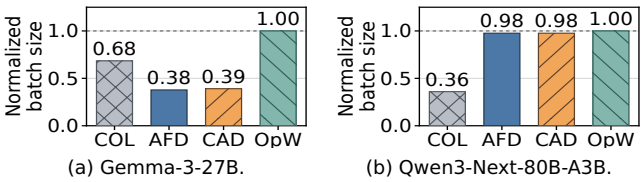


Figure 11. Mean per-GPU batch size at the relaxed TPOT SLO, normalized to OpWeave within each context and averaged across contexts. Higher is better.

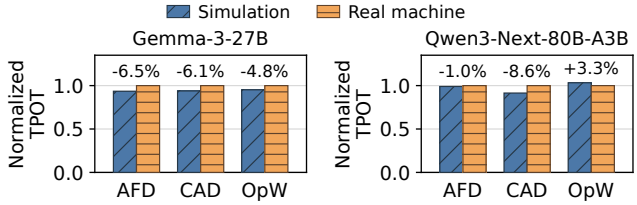


Figure 12. Simulated and measured TPOT for AFD, CAD, and OpWeave, normalized by measured TPOT. Labels show signed simulation errors.

difficult to balance. For example, at 128K context and batch size 1 on one H100, our profiles estimate 700 μs for a Qwen3-Next full-attention module versus 58 μs for linear attention, a 12× gap. OpWeave's flexible partition boundaries allow it to balance work across these layers. Moreover, AFD and CAD's high network transfer volume leaves limited latency headroom for increasing the number of microbatches to overlap work across device groups. Even under Gemma's relaxed 60-ms SLO, their selected plans still use only one microbatch. The two device groups therefore execute dependent stages sequentially, leaving one group idle while the other works.

OpWeave also alleviates the memory contention that limits its batching in colocated serving. The attention KV cache competes with model weights for GPU memory, so longer contexts reduce the maximum feasible batch size. OpWeave mitigates this constraint by decoupling GEMM batching from KV-cache memory pressure. Consistent with our theoretical analysis, Figure 11 shows that OpWeave achieves larger mean per-GPU batch sizes than colocation for both models under the relaxed SLOs. AFD and CAD similarly alleviate this constraint for Qwen3-Next, achieving batch sizes close to OpWeave's. For Gemma, however, their high communication latency forces them to use smaller batches to satisfy the TPOT SLO, resulting in lower per-GPU batch sizes even than colocation.

7.3 Simulator Fidelity

We use the simulator for large-scale heterogeneous evaluation because real-machine experiments at this scale are prohibitively expensive and heterogeneous clusters with high-speed inter-node connections are difficult to allocate on public clouds. We validate it using two H100 and two L40S GPUs for Gemma-3-27B and four of each for Qwen3-Next-80B-A3B. As shown in Figure 12, the average absolute

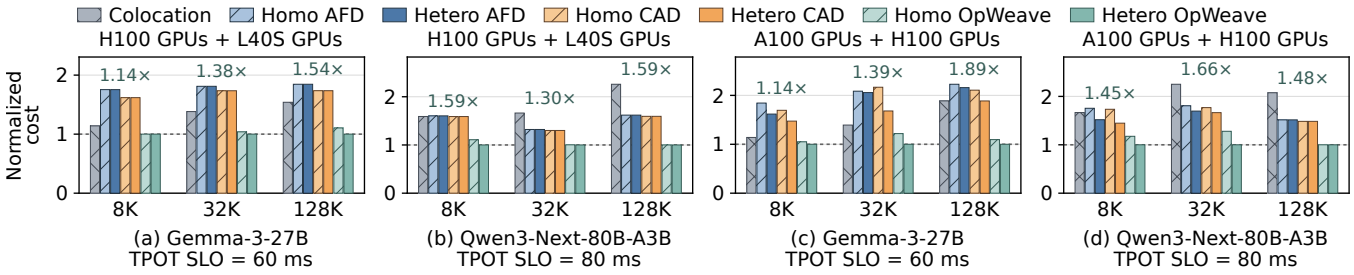


Figure 13. Simulated serving cost across context lengths and TPOT SLOs, normalized to heterogeneous OpWeave within each context. Labels show $g\times$ cheaper, where g is the best feasible non-OpWeave baseline cost divided by heterogeneous OpWeave cost.

simulation errors across AFD, CAD, and OpWeave are 5.8% for Gemma-3-27B and 4.3% for Qwen3-Next-80B-A3B.

7.4 Heterogeneous End-to-End Evaluation

We use our simulator to evaluate OpWeave and all baselines on two heterogeneous setups, H100 with L40S and H100 with A100 (Table 1), using the relaxed TPOT SLOs and all three context lengths. As shown in Figure 13, heterogeneous OpWeave achieves lower serving cost than all non-OpWeave baselines in all 12 evaluated model–hardware–context settings. On H100 with L40S, heterogeneous OpWeave is up to 1.54× cheaper for Gemma-3-27B and 1.59× cheaper for Qwen3-Next-80B-A3B than the best feasible non-OpWeave baseline. On H100 with A100, heterogeneous OpWeave is up to 1.89× and 1.66× cheaper, respectively. OpWeave thus retains its cost advantage over non-OpWeave baselines in heterogeneous settings.

Compared with homogeneous OpWeave, heterogeneous OpWeave is 1.04× cheaper on average (arithmetic mean of per-setting cost ratios across both models and contexts) and up to 1.11× cheaper on H100 with L40S. On H100 with A100, heterogeneous OpWeave is 1.14× cheaper on average and up to 1.28× cheaper. The larger benefit on H100 with A100 is consistent with the analysis in Section 3.3: a greater disparity in the GPUs' compute-to-memory-bandwidth ratios permits a higher upper bound on the idealized gain from heterogeneous assignment. Using the specifications in Table 1, the hardware ratio η is 1.42 for H100 with L40S and 1.93 for H100 with A100, yielding idealized gain bounds of 1.10× and 1.19×, respectively, under Eq. 8. The measured gains can exceed these idealized bounds because the bounds assume both deployments fully disaggregate memory-bound attention from compute-bound GEMM-based operators, which neither deployment achieves in practice.

7.5 Ablation Study and Sensitivity Study

Planner Search Ablation. This ablation quantifies the benefits of Pareto-frontier pruning and branch-and-bound for planner search. We search for Gemma-3-27B plans at 32K context under a 60-ms TPOT SLO, using 64 CPU cores for every run. Table 3 reports candidates considered, simulator invocations, and end-to-end search time. Because inexpensive memory and SLO checks discard many candidates before the costly simulator is invoked, reducing candidate count does not translate proportionally into wall-clock time. Pareto-frontier pruning reduces simulator invocations, whereas branch-and-bound removes candidates before simulation. Disabling PF increases simulator invocations by 1.85× on H100 and 3.75× on A100+H100, increasing wall-clock time by 1.16× and 1.74×, respectively. Disabling B&B instead increases considered candidates by 45.96× and 45.42× without increasing simulator invocations; these are rejected by the cheaper checks, so wall-clock time increases by only 1.27×

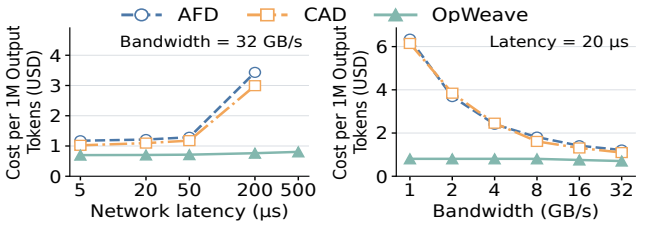


Figure 14. Network sensitivity of OpWeave, AFD and CAD.

HW	Method	Candidates	Simulations	Time (s)
H100	Full	4.78M (1.00×)	4.78M (1.00×)	228.3 (1.00×)
	No-PF	8.83M (1.85×)	8.83M (1.85×)	264.4 (1.16×)
	No-B&B	219.79M (45.96×)	4.78M (1.00×)	290.3 (1.27×)
	Neither	293.24M (61.31×)	8.83M (1.85×)	330.5 (1.45×)
A100 +H100	Full	13.89M (1.00×)	13.89M (1.00×)	446.7 (1.00×)
	No-PF	52.14M (3.75×)	52.14M (3.75×)	776.4 (1.74×)
	No-B&B	630.97M (45.42×)	13.89M (1.00×)	636.6 (1.42×)
	Neither	2.33B (167.79×)	52.14M (3.75×)	1,495.8 (3.35×)

Table 3. Planner search ablation. *Full* enables Pareto-frontier (PF) pruning and branch-and-bound (B&B); *No-PF*, *No-B&B*, and *Neither* disable the corresponding optimizations. Parentheses show degradation relative to *Full* on the same hardware. Lower is better; all methods find the same best plan.

and 1.42×. Together, they make search 1.45× faster on H100 and 3.35× on A100+H100 than using neither.

Network Sensitivity. We use simulation to evaluate the sensitivity of OpWeave, AFD, and CAD to network conditions, varying inter-node transfer latency and bandwidth for Gemma-3-27B at 8K context under a 60-ms TPOT SLO. Each method is reoptimized at every setting over H100, L40S, and mixed deployments. As shown in Figure 14, OpWeave is less sensitive than AFD and CAD to both network latency and bandwidth. This robustness follows from OpWeave's much lower network transfer volume (Figure 10(b) and (e)): the remaining transfers can be overlapped with model execution, leaving serving performance less affected by network conditions.

8 Related Work

LLM Serving Optimization. General-purpose LLM serving systems improve efficiency through continuous batching [54], paged KV-cache management [23], KV reuse and prefix caching [46, 55], prefill–decode disaggregation and scheduling [1, 44, 56], and speculative decoding [6–9, 16, 24–28, 32, 37, 42]. These techniques substantially improve LLM serving, but they do not directly address the different bottlenecks of attention and GEMMs.

Operator-Level Disaggregated Serving. MegaScale-Infer and Step-3 AFD are closest to our setting: both use a fixed two-way split that disaggregates attention from FFN, MLP, or MoE during decoding [50, 58]. FastDecode similarly separates attention and its KV cache from the dense layers, but offloads attention to distributed CPU workers rather than another GPU group [15]. Infinite-LLM distributes attention computation and KV-cache capacity across instances, rather than general operator-class placement [30]. NanoFlow also operates at operation granularity, but overlaps compute, memory, and network work within a device instead of separating attention and FFN onto distinct device groups [57]. Unlike these systems, we provide a theoretical analysis of the achievable ODS cost gains, formalize the broader serving-plan space, and show why fixed two-way splits can fail for hybrid-attention architectures such as Gemma 3, GPT-OSS, and Qwen3-Next [12, 43, 47].

Heterogeneous LLM Serving. A growing line of work serves LLMs across heterogeneous GPU resources. Helix formulates heterogeneous serving as a max-flow problem, jointly optimizing layer placement and request routing across GPU types [35]. Other systems apply asymmetric pipeline parallelism to decentralized heterogeneous or volunteer GPUs [4, 5, 17, 19–21, 29, 45, 51, 53], or exploit price-performance gaps across GPU types by routing requests to the best-suited hardware [13, 18, 34]. All of these systems partition at the transformer-layer boundary or coarser; in contrast, this paper disaggregates at the operator level, matching heterogeneous GPU types to operator classes with different resource bottlenecks inside a single layer.

9 Conclusion

This paper presented OpWeave, an end-to-end framework for heterogeneous operator-level disaggregated serving that connects theory, planning, and execution. OpWeave develops an analytical cost model that characterizes and bounds the gains of homogeneous and heterogeneous ODS over colocated serving, a regularity-aware planner that jointly optimizes operator partitioning, hardware assignment, parallelism, and batching over the partition block abstraction, and a vLLM-based distributed runtime that executes the synthesized plans across heterogeneous device groups. In our evaluation, OpWeave is up to 1.78× cheaper than the best feasible baseline on homogeneous GPUs and up to 1.89× cheaper on heterogeneous clusters, while attaining latency SLOs. We hope the analysis and abstractions in OpWeave provide a foundation for future work on operator-level disaggregation across increasingly heterogeneous models and hardware.

Acknowledgments

We used ChatGPT, Claude, and Gemini to improve the clarity and readability of the manuscript. We also used Claude and GPT to assist with code development for this work. This work was partially supported by NSF awards CNS-2211882 and CNS-2239351, a Sloan Research Fellowship, and research awards from Amazon, Cisco, Google, Jane Street, Meta, NVIDIA, Oracle, Qualcomm, and Samsung.

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. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association.

[2] Ai2. 2026. *Introducing Olmo Hybrid: Combining Transformers and Linear RNNs for Superior Scaling.* Allen Institute for Artificial Intelligence. https://allenai.org/blog/olmohybrid

[3] Kyle Aubrey and Farshad Ghodsian. 2026. Inside NVIDIA Groq 3 LPX: The Low-Latency Inference Accelerator for the NVIDIA Vera Rubin Platform. https://developer.nvidia.com/blog/inside-nvidia-groq-3-lpx-the-low-latency-inference-accelerator-for-the-nvidia-vera-rubin-platform/. NVIDIA Technical Blog. Accessed: 2026-04-16.

[4] Alexander Borzunov, Dmitry Baranchuk, Tim Dettmers, Max Ryabinin, Younes Belkada, Artem Chumachenko, Pavel Samygin, and Colin Raffel. 2023. Petals: Collaborative Inference and Fine-tuning of Large Models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*. Association for Computational Linguistics, 558–568. doi:10.18653/v1/2023.acl-demo.54

[5] Alexander Borzunov, Max Ryabinin, Artem Chumachenko, Dmitry Baranchuk, Tim Dettmers, Younes Belkada, Pavel Samygin, and Colin Raffel. 2023. Distributed Inference and Fine-tuning of Large Language Models Over the Internet. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[6] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. 2024. Medusa: Simple LLM Inference Acceleration Framework with Multiple Decoding Heads. *arXiv preprint arXiv:2401.10774* (2024).

[7] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. 2023. Accelerating Large Language Model Decoding with Speculative Sampling. *arXiv preprint arXiv:2302.01318* (2023).

[8] Siyuan Chen, Zhipeng Jia, Samira Khan, Arvind Krishnamurthy, and Phillip B. Gibbons. 2025. SLOs-Serve: Optimized Serving of Multi-SLO LLMs. *arXiv preprint arXiv:2504.08784* (2025).

[9] Zhuoming Chen, Avner May, Ruslan Svirschevski, Yuhsun Huang, Max Ryabinin, Zhihao Jia, and Beidi Chen. 2024. Sequoia: Scalable, Robust, and Hardware-aware Speculative Decoding. *arXiv preprint arXiv:2402.12374* (2024).

[10] Xianzhe Dong, Tongxuan Liu, Yuting Zeng, 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

[11] Vivek Gangasani, Andrew Smith, and Goutham Annem. 2026. Introducing Disaggregated Inference on AWS powered by llm-d. https://aws.amazon.com/blogs/machine-learning/introducing-disaggregated-inference-on-aws-powered-by-llm-d/. AWS Blog. Accessed: 2026-04-16.

[12] Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, Louis Rouillard, Thomas Mesnard, Geoffrey Cideron, Jean bastien Grill, Sabela Ramos, Edouard Yvinec, Michelle Casbon, Etienne Pot, Ivo Penchev, Gaël Liu, Francesco Visin, Kathleen Kenealy, Lucas Beyer, Xiaohai Zhai, Anton Tsitsulin, Robert Busa-Fekete, Alex Feng, Noveen Sachdeva, Benjamin Coleman, Yi Gao, Basil Mustafa, Iain Barr, Emilio Parisotto, David Tian, Matan Eyal, Colin Cherry, Jan-Thorsten Peter, Danila Sinopalnikov, Surya Bhupatiraju, Rishabh Agarwal, Mehran Kazemi, Dan Malkin, Ravin Kumar, David Vilar, Idan Brusilovsky, Jiaming Luo, Andreas Steiner, Abe Friesen, Abhanshu Sharma, Abheesht Sharma, Adi Mayrav Gilady, Adrian Goedeckemeyer, Alaa Saade, Alex Feng, Alexander Kolesnikov, Alexei Bendebury, Alvin Abdagic, Amit Vadi, András György, André Susano Pinto, Anil Das, Ankur Bapna, Antoine Miech, Antoine Yang, Antonia Paterson, Ashish Shenoy, Ayan Chakrabarti, Bilal Piot, Bo Wu, Bobak Shahriari, Bryce Petrini, Charlie Chen, Charline Le Lan, Christopher A. Choquette-Choo, CJ Carey, Cormac Brick, Daniel Deutsch, Danielle Eisenbud, Dee Cattle, Derek Cheng, Dimitris Paparas, Divyashree Shivakumar Sreepathihalli, Doug Reid, Dustin Tran, Dustin Zelle, Eric Noland, Erwin Huizenga, Eugene Kharitonov, Frederick Liu, Gagik Amirkhanyan, Glenn Cameron, Hadi Hashemi, Hanna Klimczak-Plucińska, Harman Singh, Harsh Mehta, Harshal Tushar Lehri, Hussein Hazimeh, Ian Ballantyne, Idan Szpektor, Ivan Nardini, Jean Pouget-Abadie, Jetha Chan, Joe Stanton, John Wieting, Jonathan Lai, Jordi Orbay, Joseph Fernandez, Josh Newlan, Ju yeong Ji, Jyotinder Singh, Kat Black, Kathy Yu, Kevin Hui, Kiran Vodrahalli, Klaus Greff, Linhai Qiu, Marcella Valentine, Marina Coelho, Marvin Ritter, Matt Hoffman, Matthew Watson, Mayank Chaturvedi, Michael Moynihan, Min Ma, Nabila Babar, Natasha Noy, Nathan Byrd, Nick Roy, Nikola Momchev, Nilay Chauhan, Noveen Sachdeva, Oskar Bunyan, Pankil Botarda, Paul Caron, Paul Kishan Rubenstein, Phil Culliton, Philipp Schmid, Pier Giuseppe Sessa, Pingmei Xu, Piotr Stanczyk, Pouya Tafti, Rakesh Shivanna, Renjie Wu, Renke Pan, Reza Rokni, Rob Willoughby, Rohith Vallu, Ryan Mullins, Sammy Jerome, Sara Smoot, Sertan Girgin, Shariq Iqbal, Shashir Reddy, Shruti Sheth, Siim Põder, Sijal Bhatnagar, Sindhu Raghuram Panyam, Sivan Eiger, Susan Zhang, Tianqi Liu, Trevor Yacovone, Tyler Liechty, Uday Kalra, Utku Evci, Vedant Misra, Vincent Roseberry, Vlad Feinberg, Vlad Kolesnikov, Woohyun Han, Woosuk Kwon, Xi Chen, Yinlam Chow, Yuvein Zhu, Zichuan Wei, Zoltan Egyed, Victor Cotruta, Minh Giang, Phoebe Kirk, Anand Rao, Kat Black, Nabila Babar, Jessica Lo, Erica Moreira, Luiz Gustavo Martins, Omar Sanseviero, Lucas Gonzalez, Zach Gleicher, Tris Warkentin, Vahab Mirrokni, Evan Senter, Eli Collins, Joelle Barral, Zoubin Ghahramani, Raia Hadsell, Yossi Matias, D. Sculley, Slav Petrov, Noah Fiedel, Noam Shazeer, Oriol Vinyals, Jeff Dean, Demis Hassabis, Koray Kavukcuoglu, Clement Farabet, Elena Buchatskaya, Jean-Baptiste Alayrac, Rohan Anil, Dmitry (Dima) Lepikhin, Sebastian Borgeaud, Olivier Bachem, Armand Joulin, Alek Andreev, Cassidy Hardin, Robert Dadashi, and Léonard Hussenot. 2025. Gemma 3 Technical Report. arXiv:2503.19786 [cs.CL] https://arxiv.org/abs/2503.19786

[13] Tyler Griggs, Xiaoxuan Liu, Jiaxiang Yu, Doyoung Kim, Wei-Lin Chiang, Alvin Cheung, and Ion Stoica. 2024. Mélange: Cost Efficient Large Language Model Serving by Exploiting GPU Heterogeneity. *arXiv preprint arXiv:2404.14527* (2024).

[14] Etash Guha, Ryan Marten, Sedrick Keh, Negin Raoof, Georgios Smyrnis, Hritik Bansal, Marianna Nezhurina, Jean Mercat, Trung Vu, Zayne Sprague, Ashima Suvarna, Benjamin Feuer, Liangyu Chen, Zaid Khan, Eric Frankel, Sachin Grover, Caroline Choi, Niklas Muennighoff, Shiye Su, Wanjia Zhao, John Yang, Shreyas Pimpalgaonkar, Kartik Sharma, Charlie Cheng-Jie Ji, Yichuan Deng, Sarah Pratt, Vivek Ramanujan, Jon Saad-Falcon, Jeffrey Li, Achal Dave, Alon Albalak, Kushal Arora, Blake Wulfe, Chinmay Hegde, Greg Durrett, Sewoong Oh, Mohit Bansal, Saadia Gabriel, Aditya Grover, Kai-Wei Chang, Vaishaal Shankar, Aaron Gokaslan, Mike A. Merrill, Tatsunori Hashimoto, Yejin Choi, Jenia Jitsev, Reinhard Heckel, Maheswaran Sathiamoorthy, Alexandros G. Dimakis, and Ludwig Schmidt. 2025. OpenThoughts: Data Recipes for Reasoning Models. arXiv:2506.04178 [cs.LG] https://arxiv.org/abs/2506.04178

[15] Jiaao He, Kezhao Huang, and Jidong Zhai. 2024. FASTDECODE: High-Throughput LLM Serving through Disaggregating Attention Computation. In *Proceedings of the 1st Workshop on Long-Context Foundation Models*. https://openreview.net/forum?id=GahfuPsGw2

[16] Kaiyu Huang, Hao Wu, Zhubo Shi, Han Zou, Minchen Yu, and Qingjiang Shi. 2025. AdaSpec: Adaptive Speculative Decoding for Fast, SLO-Aware Large Language Model Serving. *arXiv preprint arXiv:2503.05096* (2025).

[17] Youhe Jiang, Fangcheng Fu, Xiaozhe Yao, Taiyi Wang, Bin Cui, Ana Klimovic, and Eiko Yoneki. 2025. ThunderServe: High-performance and Cost-efficient LLM Serving in Cloud Environments. In *Proceedings of Machine Learning and Systems (MLSys)*.

[18] Youhe Jiang, Fangcheng Fu, and Eiko Yoneki. 2026. BOute: Cost-Efficient LLM Serving with Heterogeneous LLMs and GPUs via Multi-Objective Bayesian Optimization. *arXiv preprint arXiv:2602.10729* (2026).

[19] Youhe Jiang, Ran Yan, Xiaozhe Yao, Yang Zhou, Beidi Chen, and Binhang Yuan. 2024. HexGen: Generative Inference of Large Language Model over Heterogeneous Environment. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*.

[20] Youhe Jiang, Ran Yan, and Binhang Yuan. 2025. HexGen-2: Disaggregated Generative Inference of LLMs in Heterogeneous Environment. In *The Thirteenth International Conference on Learning Representations (ICLR)*.

[21] Kihyun Kim, Jinwoo Kim, James J. Kim, Dong Li, and Youngjae Kim. 2025. FlexLLM: Flexible and Cost-Efficient LLM Serving with Heterogeneous GPUs. In *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*.

[22] Kimi Team, Yu Zhang, Zongyu Lin, Xingcheng Yao, Jiaxi Hu, Fanqing Meng, Chengyin Liu, Xin Men, Songlin Yang, Zhiyuan Li, Wentao Li, Enzhe Lu, Weizhou Liu, Yanru Chen, Weixin Xu, Longhui Yu, Yejie Wang, Yu Fan, Longguang Zhong, Enming Yuan, Dehao Zhang, Yizhi Zhang, T. Y. Liu, Haiming Wang, Shengjun Fang, Weiran He,

Shaowei Liu, Yiwei Li, Jianlin Su, Jiezhong Qiu, Bo Pang, Junjie Yan, Zhejun Jiang, Weixiao Huang, Bohong Yin, Jiacheng You, Chu Wei, Zhengtao Wang, Chao Hong, Yutian Chen, Guanduo Chen, Yucheng Wang, Huabin Zheng, Feng Wang, Yibo Liu, Mengnan Dong, Zheng Zhang, Siyuan Pan, Wenhao Wu, Yuhao Wu, Longyu Guan, Jiawen Tao, Guohong Fu, Xinran Xu, Yuzhi Wang, Guokun Lai, Yuxin Wu, Xinyu Zhou, Zhilin Yang, and Yulun Du. 2025. Kimi linear: An expressive, efficient attention architecture. *arXiv preprint arXiv:2510.26692* (2025).

[23] 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. *arXiv preprint arXiv:2309.06180* (2023).

[24] Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2022. Fast Inference from Transformers via Speculative Decoding. *arXiv preprint arXiv:2211.17192* (2022).

[25] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2024. EAGLE-2: Faster Inference of Language Models with Dynamic Draft Trees. *arXiv preprint arXiv:2406.16858* (2024).

[26] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2024. EAGLE: Speculative Sampling Requires Rethinking Feature Uncertainty. *arXiv preprint arXiv:2401.15077* (2024).

[27] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2025. EAGLE-3: Scaling up Inference Acceleration of Large Language Models via Training-Time Test. *arXiv preprint arXiv:2503.01840* (2025).

[28] Zikun Li, Zhuofu Chen, Remi Delacourt, Gabriele Oliaro, Zeyu Wang, Qinghan Chen, Shuhuai Lin, April Yang, Zhihao Zhang, Zhuoming Chen, Yi-Hsiang Lai, Xinhao Cheng, Xupeng Miao, and Zhihao Jia. 2026. AdaServe: Accelerating Multi-SLO LLM Serving with SLO-Customized Speculative Decoding. In *Proceedings of the 21st European Conference on Computer Systems (EuroSys '26)*. ACM, New York, NY, USA, 36–54. doi:10.1145/3767295.3769315

[29] Zonghang Li, Wenjiao Feng, Mohsen Guizani, and Hongfang Yu. 2024. TPI-LLM: Serving 70B-scale LLMs Efficiently on Low-resource Edge Devices. *arXiv preprint arXiv:2410.00531* (2024).

[30] Bin Lin, Chen Zhang, Tao Peng, Hanyu Zhao, Wencong Xiao, Minmin Sun, Anmin Liu, Zhipeng Zhang, Lanbo Li, Xiafei Qiu, Shen Li, Zhigang Ji, Tao Xie, Yong Li, and Wei Lin. 2024. Infinite-LLM: Efficient LLM Service for Long Context with DistAttention and Distributed KVCache. *arXiv preprint arXiv:2401.02669* (2024).

[31] Guowei Liu, Hongming Li, Yaning Guo, Yongxi Lyu, Mo Zhou, Yi Liu, Zhaogeng Li, and Yanpeng Wang. 2026. Revealing the Challenges of Attention-FFN Disaggregation for Modern MoE Models and Hardware Systems. *arXiv preprint arXiv:2602.09721* (2026). https://arxiv.org/abs/2602.09721

[32] Xiaoxuan Liu, Jongseok Park, Langxiang Hu, Woosuk Kwon, Zhuohan Li, Chen Zhang, Kuntai Du, Xiangxi Mo, Kaichao You, Alvin Cheung, Zhijie Deng, Ion Stoica, and Hao Zhang. 2024. TurboSpec: Closed-loop Speculation Control System for Optimizing LLM Serving Goodput. *arXiv preprint arXiv:2406.14066* (2024).

[33] Mark Lohmeyer and Drew Bradstock. 2025. GKE Inference Gateway and Quickstart are GA. https://cloud.google.com/blog/products/ai-machine-learning/gke-inference-gateway-and-quickstart-are-ga. Google Cloud Blog. Accessed: 2026-04-16.

[34] Yixuan Mei, Zikun Li, Zixuan Chen, Shiqi Pan, Mengdi Wu, Xupeng Miao, Zhihao Jia, and K. V. Rashmi. 2026. Coral: Cost-Efficient Multi-LLM Serving over Heterogeneous Cloud GPUs. *arXiv preprint arXiv:2605.04357* (2026).

[35] Yixuan Mei, Yonghao Zhuang, Xupeng Miao, Juncheng Yang, Zhihao Jia, and Rashmi Vinayak. 2024. Helix: Serving Large Language Models over Heterogeneous GPUs and Network via Max-Flow. *arXiv preprint arXiv:2406.01566* (2024).

[36] Nicholas Merizzi, Chris Thomas, and Ed Burns. 2025. The AI Infrastructure Reckoning: Optimizing Compute Strategy in the Age of Inference Economics. https://www.deloitte.com/us/en/insights/topics/technology-management/tech-trends/2026/ai-infrastructure-compute-strategy.html. Deloitte Insights. Accessed: 2026-04-15.

[37] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. 2024. SpecInfer: Accelerating Large Language Model Serving with Tree-based Speculative Inference and Verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. ACM, New York, NY, USA, 932–949.

[38] Jesse Noffsinger, Mark Patel, Pankaj Sachdeva, Arjita Bhan, Haley Chang, and Maria Goodpaster. 2025. The Cost of Compute: A $7 Trillion Race to Scale Data Centers. https://www.mckinsey.com/industries/technology-media-and-telecommunications/our-insights/the-cost-of-compute-a-7-trillion-dollar-race-to-scale-data-centers. McKinsey & Company. Accessed: 2026-04-18.

[39] NVIDIA. 2026. A100 Tensor Core GPU: Specifications. https://www.nvidia.com/en-us/data-center/a100/. Accessed: 2026-09-07.

[40] NVIDIA. 2026. H100 Tensor Core GPU: Product Specifications. https://www.nvidia.com/en-sg/data-center/h100/. Accessed: 2026-09-07.

[41] NVIDIA. 2026. L40S GPU Specifications. https://www.nvidia.com/en-us/data-center/l40s/. Accessed: 2026-09-07.

[42] Gabriele Oliaro, Zhihao Jia, Daniel Campos, and Aurick Qiao. 2024. SuffixDecoding: Extreme Speculative Decoding for Emerging AI Applications. *arXiv preprint arXiv:2411.04975* (2024).

[43] OpenAI, Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K. Arora, Yu Bai, Bowen Baker, Haiming Bao, Boaz Barak, Ally Bennett, Tyler Bertao, Nivedita Brett, Eugene Brevdo, Greg Brockman, Sebastien Bubeck, Che Chang, Kai Chen, Mark Chen, Enoch Cheung, Aidan Clark, Dan Cook, Marat Dukhan, Casey Dvorak, Kevin Fives, Vlad Fomenko, Timur Garipov, Kristian Georgiev, Mia Glaese, Tarun Gogineni, Adam Goucher, Lukas Gross, Katia Gil Guzman, John Hallman, Jackie Hehir, Johannes Heidecke, Alec Helyar, Haitang Hu, Romain Huet, Jacob Huh, Saachi Jain, Zach Johnson, Chris Koch, Irina Kofman, Dominik Kundel, Jason Kwon, Volodymyr Kyrylov, Elaine Ya Le, Guillaume Leclerc, James Park Lennon, Scott Lessans, Mario Lezcano-Casado, Yuanzhi Li, Zhuohan Li, Ji Lin, Jordan Liss, Lily (Xiaoxuan) Liu, Jiancheng Liu, Kevin Lu, Chris Lu, Zoran Martinovic, Lindsay McCallum, Josh McGrath, Scott McKinney, Aidan McLaughlin, Song Mei, Steve Mostovoy, Tong Mu, Gideon Myles, Alexander Neitz, Alex Nichol, Jakub Pachocki, Alex Paino, Dana Palmie, Ashley Pantuliano, Giambattista Parascandolo, Jongsoo Park, Leher Pathak, Carolina Paz, Ludovic Peran, Dmitry Pimenov, Michelle Pokrass, Elizabeth Proehl, Huida Qiu, Gaby Raila, Filippo Raso, Hongyu Ren, Kimmy Richardson, David Robinson, Bob Rotsted, Hadi Salman, Suvansh Sanjeev, Max Schwarzer, D. Sculley, Harshit Sikchi, Kendal Simon, Karan Singhal, Yang Song, Dane Stuckey, Zhiqing Sun, Philippe Tillet, Sam Toizer, Foivos Tsimpourlas, Nikhil Vyas, Eric Wallace, Xin Wang, Miles Wang, Olivia Watkins, Kevin Weil, Amy Wendling, Kevin Whinnery, Cedric Whitney, Hannah Wong, Lin Yang, Yu Yang, Michihiro Yasunaga, Kristen Ying, Wojciech Zaremba, Wenting Zhan, Cyril Zhang, Brian Zhang, Eddie Zhang, and Shengjia Zhao. 2025. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925* (2025).

[44] 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.

[45] You Peng, Youhe Jiang, Wenqi Jiang, Chen Wang, and Binhang Yuan. 2025. HEXGEN-FLOW: Optimizing LLM Inference Request Scheduling for Agentic Text-to-SQL. *arXiv preprint arXiv:2505.05286* (2025).

[46] 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* (2024).

[47] Qwen Team. 2025. Qwen3-Next: Towards Ultimate Training & Inference Efficiency. https://qwen.ai/blog?id=4074cca80393150c248e508aa62983f9cb7d27cd. Qwen blog, September 10, 2025.

[48] 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)*. 55740–55756. https://openreview.net/forum?id=n7VLFYNoCb

[49] Chendong Song, Meixuan Wang, Hang Zhou, Hong Liang, Yuan Lyu, Zixi Chen, Yuwei Fan, and Zijie Zhou. 2026. Analytical Provisioning for Attention-FFN Disaggregated LLM Serving under Stochastic Workloads. *arXiv preprint arXiv:2601.21351* (2026). https://arxiv.org/abs/2601.21351

[50] StepFun, Bin Wang, Bojun Wang, Changyi Wan, Guanzhe Huang, Hanpeng Hu, Haonan Jia, Hao Nie, Mingliang Li, Nuo Chen, Siyu Chen, Song Yuan, Wuxun Xie, Xiaoniu Song, Xing Chen, Xingping Yang, Xuelin Zhang, Yanbo Yu, Yaoyu Wang, Yibo Zhu, Yimin Jiang, Yu Zhou, Yuanwei Lu, Houyi Li, Jingcheng Hu, Ka Man Lo, Ailin Huang, Binxing Jiao, Bo Li, Boyu Chen, Changxin Miao, Chang Lou, Chen Hu, Chen Xu, Chenfeng Yu, Chengyuan Yao, Daokuan Lv, Dapeng Shi, Deshan Sun, Ding Huang, Dingyuan Hu, Dongqing Pang, Enle Liu, Fajie Zhang, Fanqi Wan, Gulin Yan, Han Zhang, Han Zhou, Hanghao Wu, Hangyu Guo, Hanqi Chen, Hanshan Zhang, Hao Wu, Haocheng Zhang, Haolong Yan, Haoran Lv, Haoran Wei, Hebin Zhou, Heng Wang, Heng Wang, Hongxin Li, Hongyu Zhou, Hongyuan Wang, Huiyong Guo, Jia Wang, Jiahao Gong, Jialing Xie, Jian Zhou, Jianjian Sun, Jiaoren Wu, Jiaran Zhang, Jiayu Liu, Jie Cheng, Jie Luo, Jie Yan, Jie Yang, Jieyi Hou, Jinguang Zhang, Jinlan Cao, Jisheng Yin, Junfeng Liu, Junhao Huang, Junzhe Lin, Kaijun Tan, Kaixiang Li, Kang An, Kangheng Lin, Kenkun Liu, Lei Yang, Liang Zhao, Liangyu Chen, Lieyu Shi, Liguo Tan, Lin Lin, Lin Zhang, Lina Chen, Liwen Huang, Liying Shi, Longlong Gu, Mei Chen, Mengqiang Ren, Ming Li, Mingzhe Chen, Na Wang, Nan Wu, Qi Han, Qian Zhao, Qiang Zhang, Qianni Liu, Qiaohui Chen, Qiling Wu, Qinglin He, Qinyuan Tan, Qiufeng Wang, Qiuping Wu, Qiuyan Liang, Quan Sun, Rui Li, Ruihang Miao, Ruosi Wan, Ruyan Guo, Shangwu Zhong, Shaoliang Pang, Shengjie Fan, Shijie Shang, Shilei Jiang, Shiliang Yang, Shiming Hao, Shuli Gao, Siming Huang, Siqi Liu, Tiancheng Cao, Tianhao Cheng, Tianhao Peng, Wang You, Wei Ji, Wen Sun, Wenjin Deng, Wenqing He, Wenzhen Zheng, Xi Chen, Xiangwen Kong, Xianzhen Luo, Xiaobo Yang, Xiaojia Liu, Xiaoxiao Ren, Xin Han, Xin Li, Xin Wu, Xu Zhao, Yanan Wei, Yang Li, Yangguang Li, Yangshijie Xu, Yanming Xu, Yaqiang Shi, Yeqing Shen, Yi Yang, Yifei Yang, Yifeng Gong, Yihan Chen, Yijing Yang, Yinmin Zhang, Yizhuang Zhou, Yuanhao Ding, Yuantao Fan, Yuanzhen Yang, Yuchu Luo, Yue Peng, Yufan Lu, Yuhang Deng, Yuhe Yin, Yujie Liu, Yukun Chen, Yuling Zhao, Yun Mou, Yunlong Li, Yunzhou Ju, Yusheng Li, Yuxiang Yang, Yuxiang Zhang, Yuyang Chen, Zejia Weng, Zhe Xie, Zheng Ge, Zheng Gong, Zhenyi Lu, Zhewei Huang, Zhichao Chang, Zhiguo Huang, Zhirui Wang, Zidong Yang, Zili Wang, Ziqi Wang, Zixin Zhang, Binxing Jiao, Daxin Jiang, Heung-Yeung Shum, and Xiangyu Zhang. 2025. Step-3 is Large yet Affordable: Model-system Co-design for Cost-effective Decoding. *arXiv preprint arXiv:2507.19427* (2025).

[51] Chris Tong, Youhe Jiang, Gufeng Chen, Tianyi Zhao, Sibian Lu, Wenjie Qu, Eric Yang, Lynn Ai, and Binhang Yuan. 2025. Parallax: Efficient LLM Inference Service over Decentralized Environment. *arXiv preprint arXiv:2509.26182* (2025).

[52] Tim Tully, Joff Redfern, Deedy Das, and Derek Xiao. 2025. 2025 Mid-Year LLM Market Update: Foundation Model Landscape & Economics. https://menlovc.com/perspective/2025-mid-year-llm-market-update/. Menlo Ventures. Accessed: 2026-04-15.

[53] Linyu Wu, Xiaoyuan Liu, Tianneng Shi, Zhe Ye, and Dawn Song. 2025. DeServe: Towards Affordable Offline LLM Inference via Decentralization. *arXiv preprint arXiv:2501.14784* (2025).

[54] 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, 521–538.

[55] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2023. SGLang: Efficient Execution of Structured Language Model Programs. *arXiv preprint arXiv:2312.07104* (2023).

[56] 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. *arXiv preprint arXiv:2401.09670* (2024).

[57] Kan Zhu, Yufei Gao, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Tian Tang, Qinyu Xu, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, Ziren Wang, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. 2024. NanoFlow: Towards Optimal Large Language Model Serving Throughput. *arXiv preprint arXiv:2408.12757* (2024).

[58] Ruidong Zhu, Ziheng Jiang, Chao Jin, Peng Wu, Cesar A. Stuardo, Dongyang Wang, Xinlei Zhang, Huaping Zhou, Haoran Wei, Yang Cheng, Jianzhe Xiao, Xinyi Zhang, Lingjun Liu, Haibin Lin, Li-Wen Chang, Jianxi Ye, Xiao Yu, Xuanzhe Liu, Xin Jin, and Xin Liu. 2025. MegaScale-Infer: Serving Mixture-of-Experts at Scale with Disaggregated Expert Parallelism. *arXiv preprint arXiv:2504.02263* (2025).

A Analytical Model and Proofs for Section 3

This appendix section collects the analytical model, proofs, and CAD feasibility details used by Section 3.

A.1 Detailed analytical model

We now formalize the analytical abstraction used in Section 3.1. We model one decode step of an LLM as the sequential execution of an ordered operator sequence

$$\mathbf{o} = (o_1, o_2, \ldots, o_{N_{\text{model}}}),$$

where each o_i denotes one computational kernel invoked during the decode step. This abstraction follows the execution pattern of current serving engines, which invoke the model's kernels in a fixed topological order during each decoding iteration.

For an operator o_i executed at batch size b, let $D_i(b)$ denote the total number of bytes loaded from memory and let $W_i(b)$ denote the total number of floating-point operations. We characterize each GPU type by four quantities: unit-time monetary cost c, memory bandwidth β, memory capacity M, and peak compute throughput F. Under the exclusive-occupancy assumption, the monetary cost of operator o_i is

$$\text{Cost}(o_i) = T(o_i) \cdot c,$$

where $T(o_i)$ is the execution latency of the operator. We model this latency using a roofline-style expression:

$$T(o_i) = \max\left(\frac{D_i(b)}{\beta}, \frac{W_i(b)}{F}\right).$$

The first term is the memory-load time and the second term is the compute time, so the maximum captures whether the operator is memory-bound or compute-bound on the target GPU.

The cost of a full decode step is then the sum of the costs of all operators in the sequence:

$$\text{Cost}_{\text{step}} = \sum_{i=1}^{N_{\text{model}}} \text{Cost}(o_i).$$

Since one decode step produces one token per request, the GPU cost per token is obtained by dividing $\text{Cost}_{\text{step}}$ by the batch size b.

Throughout the theoretical analysis, we intentionally focus on GPU-side cost. Although a disaggregated pipeline introduces activation transfers between device groups, the analysis studies the idealized steady-state regime in which these transfers are overlapped with computation through microbatching and pipelining. Under this assumption, communication does not lie on the throughput-critical path and is therefore omitted from the analytical objective.

A.2 Execution-regime assumptions

This subsection formalizes the execution-regime assumptions used in Section 3.1. Under the roofline model above, an operator is *memory-bound* when

$$\frac{D_i(b)}{\beta} \geq \frac{W_i(b)}{F},$$

and *compute-bound* when the reverse inequality holds.

We assume that decode-phase attention lies in the memory-bound regime. During decoding, each request contributes only one query token while the attention operator must load the full KV cache associated with the prior context, which keeps arithmetic intensity low and makes latency primarily determined by memory traffic. Formally, for attention operators o_i^{attn}, we assume

$$\frac{D_i^{\text{attn}}(b)}{\beta} \geq \frac{W_i^{\text{attn}}(b)}{F}.$$

For GEMM-dominated operators, we assume that there exists a batch-size threshold b^* such that these operators become compute-bound once the batch size is sufficiently large. Formally, for GEMM operators o_i^{gemm}, we assume that for all $b \geq b^*$,

$$\frac{W_i^{\text{gemm}}(b)}{F} \geq \frac{D_i^{\text{gemm}}(b)}{\beta}.$$

The threshold b^* marks the transition at which arithmetic time overtakes memory-load time, so beyond this point further execution is limited by compute throughput rather than bandwidth.

The role of disaggregation is to make this compute-bound GEMM regime attainable. In colocated serving, GEMM operators share device memory with the KV cache, so long contexts reduce the maximum feasible batch size and can keep GEMM in a memory-bound regime. Under operator-level disaggregation, the compute side no longer needs to reserve memory for the KV cache, and requests forwarded from multiple attention-side replicas can be aggregated on the GEMM side. This decoupling allows the GEMM side to sustain a larger effective batch size and thus operate in the compute-bound regime assumed in the main-text analysis.

A.3 Derivation and interpretation of the final cost expressions

This subsection derives the cost expressions used in Section 3.1 from the operator-level roofline model.

Operator-class aggregates. Let O_{attn} denote the set of attention operators executed in one decode step and let O_{gemm} denote the set of GEMM-dominated operators, where the

latter includes the linear projections, FFN, and MoE computation. We define the aggregate quantities

$$\begin{aligned} D^{\text{attn}}(b) &= \sum_{o_i \in O_{\text{attn}}} D_i(b), \\ D^{\text{gemm}}(b) &= \sum_{o_i \in O_{\text{gemm}}} D_i(b), \\ W^{\text{gemm}}(b) &= \sum_{o_i \in O_{\text{gemm}}} W_i(b). \end{aligned}$$

These aggregate terms collect, respectively, the total attention-side memory traffic, the total GEMM-side memory traffic, and the total GEMM-side floating-point work for a batch of size b in one decode step.

Substituting these class aggregates into the roofline cost model of Appendix A.1 and dividing by the batch size yields the per-token costs as follows.

Homogeneous disaggregated serving. In homogeneous ODS, attention and GEMM-dominated operators are placed on separate device groups of the same GPU type (c, β, M, F). Under the execution-regime assumptions of Section 3.1, decode-phase attention is memory-bound while GEMM-dominated operators are compute-bound. Their class-level costs therefore reduce to

$$\text{Cost}^{\text{attn}}_{\text{step}} = \frac{D^{\text{attn}}(b)}{\beta}\, c, \qquad \text{Cost}^{\text{gemm}}_{\text{step}} = \frac{W^{\text{gemm}}(b)}{F}\, c.$$

Dividing the sum by b yields

$$\text{CPT}^{\text{hom}} = \frac{c}{b}\left(\frac{D^{\text{attn}}(b)}{\beta} + \frac{W^{\text{gemm}}(b)}{F}\right).$$

Heterogeneous disaggregated serving. In heterogeneous ODS, the attention side and the GEMM side may run on different GPU types. If attention runs on GPU type 1 with parameters (c_1, β_1, M_1, F_1) and GEMM-dominated operators run on GPU type 2 with parameters (c_2, β_2, M_2, F_2), the same regime assumptions give

$$\text{Cost}^{\text{attn}}_{\text{step}} = \frac{D^{\text{attn}}(b)}{\beta_1}\, c_1, \qquad \text{Cost}^{\text{gemm}}_{\text{step}} = \frac{W^{\text{gemm}}(b)}{F_2}\, c_2,$$

and thus

$$\text{CPT}^{\text{het}}_{1,2} = \frac{1}{b}\left(\frac{D^{\text{attn}}(b)}{\beta_1} c_1 + \frac{W^{\text{gemm}}(b)}{F_2} c_2\right).$$

The cost-efficiency metrics α_j and γ_j defined in Section 3.1 measure monetary cost per byte of memory traffic and per FLOP, respectively. Heterogeneous specialization allows attention to use the GPU type with the lower α_j and GEMM-dominated operators to use the type with the lower γ_j.

Colocated serving. In colocated serving, all operators share the same device memory, so the feasible batch size is constrained by both model weights and the KV cache. Let M_{weights} denote the memory occupied by model weights and let $m_{\text{kv}}(s)$ denote the KV-cache memory required per request at average sequence length s. Neglecting activation memory for simplicity, the maximum feasible batch size is

$$b_{\max}(s) = \left\lfloor \frac{M - M_{\text{weights}}}{m_{\text{kv}}(s)} \right\rfloor.$$

As s grows, $m_{\text{kv}}(s)$ increases and the feasible batch size shrinks.

Attention remains memory-bound under colocation, so its contribution is still

$$\frac{D^{\text{attn}}(b)}{\beta}\, c.$$

For GEMM-dominated operators, however, the smaller feasible batch size may prevent the compute side from reaching the compute-bound regime. We therefore retain the full roofline expression,

$$\text{Cost}^{\text{gemm}}_{\text{step}} = \max\left(\frac{D^{\text{gemm}}(b)}{\beta}, \frac{W^{\text{gemm}}(b)}{F}\right) c,$$

which gives, for $1 \le b \le b_{\max}(s)$,

$$\text{CPT}^{\text{coloc}}(s, b) = \frac{c}{b}\left(\frac{D^{\text{attn}}(b)}{\beta} + \max\left(\frac{D^{\text{gemm}}(b)}{\beta}, \frac{W^{\text{gemm}}(b)}{F}\right)\right).$$

Intuition for the small-batch regime. The max term in the colocated expression is important because GEMM efficiency depends on batch size. When the feasible batch size remains large enough, GEMM can stay compute-bound and its contribution is governed by $W^{\text{gemm}}(b)/F$. But once KV-cache growth drives the feasible batch size down, GEMM may enter a small-batch regime in which loading weights dominates the arithmetic work. In that case,

$$\frac{D^{\text{gemm}}(b)}{\beta} > \frac{W^{\text{gemm}}(b)}{F},$$

so GEMM becomes memory-bound and the GPU's compute capability is underutilized. This is precisely the inefficiency that operator-level disaggregation removes: by decoupling the compute side from KV-cache memory pressure, disaggregation allows GEMM-dominated operators to recover the larger effective batch sizes needed to operate in the compute-bound regime.

Conditional achievability via CAD. The idealized disaggregated cost expressions above are attainable by the Core-Attention Disaggregation (CAD) partition under pipeline balance. CAD assigns the core attention kernels to the attention device group and assigns the remaining GEMM-dominated operators, including projections, FFN, and MoE computation, to the rest-of-model device group. Under the regime assumptions in Appendix A.2, the attention side therefore incurs bandwidth cost while the rest-of-model side incurs compute cost. If the two stages are provisioned so that their steady-state latencies match and inter-stage communication

is overlapped by microbatching, the realized per-token GPU cost is exactly the homogeneous or heterogeneous ODS cost expression derived above.

A.4 Scaling approximations and GEMM threshold

This subsection formalizes the approximations used in Section 3.1 to obtain closed-form analytical results.

Attention-side memory traffic. During decoding, each request contributes one newly generated token, but attention must load the KV cache associated with that request's prior context. As a result, the total attention-side memory traffic grows linearly with the batch size and with the average sequence length. We therefore approximate

$$D^{\text{attn}}(b) \approx b\, m_{\text{kv}}(s),$$

where $m_{\text{kv}}(s)$ denotes the KV-cache memory required per request at average sequence length s.

GEMM-side floating-point work. We approximate the total GEMM work in one decode step as

$$W^{\text{gemm}}(b) \approx 2bP_{\text{act}},$$

where P_{act} is the effective number of GEMM-side parameters activated by one request on the modeled device group after any model-parallel sharding. For dense models, this includes all GEMM-side parameters assigned to the device group. For MoE models, it includes shared dense parameters and only the routed expert parameters used by the request, rather than all resident expert parameters. The factor of 2 reflects one multiply-add per active parameter per request. This approximation captures the fact that GEMM-side arithmetic scales linearly with the number of concurrently processed requests.

GEMM-side memory traffic. For GEMM-dominated operators, we approximate the total memory traffic by the model-weight footprint:

$$D^{\text{gemm}}(b) \approx M_{\text{weights}},$$

where M_{weights} is the resident footprint of all model weights assigned to the modeled device group after sharding. In an MoE model, inactive experts do not contribute to P_{act} for an individual request, but their resident weights still contribute to M_{weights}. The approximation $D^{\text{gemm}}(b) \approx M_{\text{weights}}$ corresponds to a sufficiently large batch whose routed tokens collectively access and amortize the resident expert-weight shards; the runtime planner uses measured operator costs rather than relying on this approximation.

Continuous form of the colocated batch-size limit. In the main text, the exact colocated feasible batch size is defined as

$$b_{\max}(s) = \left\lfloor \frac{M - M_{\text{weights}}}{m_{\text{kv}}(s)} \right\rfloor.$$

For the closed-form analysis, we use the continuous relaxation

$$\bar{b}(s) = \frac{M - M_{\text{weights}}}{m_{\text{kv}}(s)}.$$

Dropping the floor changes the feasible batch size by at most one request.

Derivation of the GEMM threshold. The threshold b^* is defined as the batch size at which GEMM transitions from memory-bound to compute-bound execution. Under the approximations above, the GEMM memory-load time is

$$\frac{D^{\text{gemm}}(b)}{\beta} \approx \frac{M_{\text{weights}}}{\beta},$$

while the GEMM compute time is

$$\frac{W^{\text{gemm}}(b)}{F} \approx \frac{2bP_{\text{act}}}{F}.$$

Equating these two terms gives the transition point:

$$\frac{M_{\text{weights}}}{\beta} = \frac{2bP_{\text{act}}}{F},$$

and therefore

$$b^* = \frac{M_{\text{weights}} F}{2P_{\text{act}}\beta}.$$

Interpretation. When $b \geq b^*$, GEMM compute time exceeds weight-loading time, so GEMM operates in the compute-bound regime. When $b < b^*$, the opposite holds and GEMM is memory-bound. This threshold is central to the later gain analysis: homogeneous ODS begins to improve over colocation precisely when KV-cache pressure pushes the colocated feasible batch size below this compute-bound threshold.

Scope of the approximation. These approximations are introduced only to obtain interpretable closed-form characterizations of when ODS helps and what bounds its gain. They are not intended as a full performance model of the runtime system; the later planner and end-to-end evaluation use richer model- and hardware-specific information.

A.5 Proofs for Section 3

This appendix provides complete proofs for the theorems stated in Section 3. We use the same notation and scaling approximations introduced in Section 3.1: $D^{\text{attn}}(b) = b \cdot m_{\text{kv}}(s)$, $W^{\text{gemm}}(b) = 2bP_{\text{act}}$, $D^{\text{gemm}}(b) = M_{\text{weights}}$, and the continuous relaxation $\bar{b}(s) = (M - M_{\text{weights}})/m_{\text{kv}}(s)$. For the homogeneous comparison, we restrict attention to sequence lengths with $b_{\max}(s) \geq 1$ and assume $m_{\text{kv}}(s)$ is strictly increasing on this feasible domain, as stated in Section 3.2.

A.5.1 Proof of Theorem 3.1: Homogeneous Disaggregation vs. Colocation.

Step 1: Simplify the disaggregated CPT. Suppressing the fixed GPU-type index and expanding $\alpha = c/\beta$ and $\gamma = c/F$ in Eq. 1 gives:

$$\mathrm{CPT}^{\mathrm{hom}} = \frac{c}{b}\left(\frac{b \cdot m_{\mathrm{kv}}(s)}{\beta} + \frac{2bP_{\mathrm{act}}}{F}\right) = c\left(\frac{m_{\mathrm{kv}}(s)}{\beta} + \frac{2P_{\mathrm{act}}}{F}\right). \tag{11}$$

The batch size b cancels because both the attention memory traffic and the GEMM compute scale linearly with b. The disaggregated CPT depends on s only through $m_{\mathrm{kv}}(s)$.

Step 2: Simplify the colocated CPT. For an arbitrary local batch size b, the colocated cost derived in Appendix A.3 becomes:

$$\begin{aligned}\mathrm{CPT}^{\mathrm{coloc}}(s,b) &= \frac{c}{b}\left(\frac{b \cdot m_{\mathrm{kv}}(s)}{\beta} + \max\left(\frac{M_{\mathrm{weights}}}{\beta},\ \frac{2bP_{\mathrm{act}}}{F}\right)\right)\\ &= c\left(\frac{m_{\mathrm{kv}}(s)}{\beta} + \frac{1}{b}\max\left(\frac{M_{\mathrm{weights}}}{\beta},\ \frac{2bP_{\mathrm{act}}}{F}\right)\right).\end{aligned} \tag{12}$$

Here b is a local batch-size variable; Section 4 distinguishes the global, per-replica, and microbatch quantities used by the planner. The cost is non-increasing in b, so its minimum over $1 \le b \le \bar{b}(s)$ is attained at $b = \bar{b}(s)$, yielding $\mathrm{CPT}^{\mathrm{coloc}}(s)$ in Eq. 4. The two GEMM regimes yield:

Compute-bound GEMM ($\bar{b}(s) \ge b^*$):

$$\mathrm{CPT}^{\mathrm{coloc}} = c\left(\frac{m_{\mathrm{kv}}(s)}{\beta} + \frac{2P_{\mathrm{act}}}{F}\right). \tag{13}$$

Memory-bound GEMM ($\bar{b}(s) < b^*$):

$$\mathrm{CPT}^{\mathrm{coloc}} = c\left(\frac{m_{\mathrm{kv}}(s)}{\beta} + \frac{M_{\mathrm{weights}}}{\bar{b}(s) \cdot \beta}\right). \tag{14}$$

Step 3: Case 1: Compute-bound GEMM ($\bar{b}(s) \ge b^*$). When the colocated batch size is large enough for GEMM to saturate compute, comparing Eq. 11 and Eq. 13 gives

$$\mathrm{CPT}^{\mathrm{coloc}} = c\left(\frac{m_{\mathrm{kv}}(s)}{\beta} + \frac{2P_{\mathrm{act}}}{F}\right) = \mathrm{CPT}^{\mathrm{hom}}.$$

Therefore $G_{\mathrm{hom}}(s) = 1$. This establishes claim 1: when the batch size is large enough for GEMM to fully utilize compute under colocation, there is no efficiency gap for disaggregation to exploit.

Step 4: Case 2: Memory-bound GEMM ($\bar{b}(s) < b^*$). When KV-cache pressure pushes $\bar{b}(s)$ below b^*, GEMM under colocation becomes memory-bound while disaggregation maintains compute-bound GEMM.

Step 4a: Rewrite the colocated CPT. Substituting $\bar{b}(s) = (M - M_{\mathrm{weights}})/m_{\mathrm{kv}}(s)$ into Eq. 14:

$$\begin{aligned}\mathrm{CPT}^{\mathrm{coloc}} &= c\left(\frac{m_{\mathrm{kv}}(s)}{\beta} + \frac{M_{\mathrm{weights}} \cdot m_{\mathrm{kv}}(s)}{(M - M_{\mathrm{weights}}) \cdot \beta}\right)\\ &= \frac{c \cdot m_{\mathrm{kv}}(s)}{\beta}\left(1 + \frac{M_{\mathrm{weights}}}{M - M_{\mathrm{weights}}}\right)\\ &= \frac{c \cdot m_{\mathrm{kv}}(s)}{\beta} \cdot \frac{M}{M - M_{\mathrm{weights}}}.\end{aligned} \tag{15}$$

Step 4b: Cost gain expression. Dividing Eq. 15 by Eq. 11:

$$G_{\mathrm{hom}}(s) = \frac{m_{\mathrm{kv}}(s)/\beta}{m_{\mathrm{kv}}(s)/\beta + 2P_{\mathrm{act}}/F} \cdot \frac{M}{M - M_{\mathrm{weights}}}. \tag{16}$$

Step 4c: Monotonicity. Let $x = m_{\mathrm{kv}}(s)/\beta$, which increases with s since $m_{\mathrm{kv}}(s)$ is increasing in s. Define $A = M/(M - M_{\mathrm{weights}}) > 1$ (since $M_{\mathrm{weights}} > 0$) and $\kappa = 2P_{\mathrm{act}}/F > 0$. Then

$$G_{\mathrm{hom}} = \frac{Ax}{x + \kappa}, \qquad \frac{dG_{\mathrm{hom}}}{dx} = \frac{A\kappa}{(x + \kappa)^2} > 0. \tag{17}$$

Since $dG_{\mathrm{hom}}/dx > 0$ for all $x > 0$, the cost gain G_{hom} is strictly increasing in x and therefore in s. This establishes claim 2.

Step 4d: Upper bound. For any feasible s in the memory-bound regime, $G_{\mathrm{hom}}(s) < A$ because $Ax/(x + \kappa) < A$ for every finite $x > 0$. Combining this fact with the compute-bound case, where $G_{\mathrm{hom}}(s) = 1 \le A$, gives the following bound for every feasible s with $b_{\max}(s) \ge 1$:

$$1 \ \le\ G_{\mathrm{hom}}(s)\ <\ \frac{M}{M - M_{\mathrm{weights}}}.$$

This establishes claim 3 and completes the proof. □

Summary. The cost gain is $G_{\mathrm{hom}}(s) = 1$ when $\bar{b}(s) \ge b^*$. When $\bar{b}(s) < b^*$, it is

$$G_{\mathrm{hom}}(s) = \frac{m_{\mathrm{kv}}(s)/\beta}{m_{\mathrm{kv}}(s)/\beta + 2P_{\mathrm{act}}/F} \cdot \frac{M}{M - M_{\mathrm{weights}}}. \tag{18}$$

Here, equality holds in the compute-bound case, while the displayed formula applies in the memory-bound case. Over the feasible memory-bound domain, G_{hom} is strictly increasing in s and bounded above by $M/(M - M_{\mathrm{weights}})$.

A.5.2 Proof of Theorem 3.2: Heterogeneous vs. Homogeneous Disaggregation.

Let $\mathcal{I}$ be the feasible interval from Theorem 3.2, and let $x = m_{\mathrm{kv}}(s)$. We first establish the shape of the analytically extended gain as a function of $x \ge 0$, then restrict it to $x \in m_{\mathrm{kv}}(\mathcal{I})$.

Step 1: $G_{\mathrm{het}}(s) \ge 1$ (Claims 1 and 2). For each GPU type j, the heterogeneous CPT satisfies

$$\begin{aligned}\mathrm{CPT}^{\mathrm{het}}_{\star} &= \min(\alpha_1, \alpha_2) \cdot m_{\mathrm{kv}}(s) + \min(\gamma_1, \gamma_2) \cdot 2P_{\mathrm{act}}\\ &\le\ \alpha_j \cdot m_{\mathrm{kv}}(s) + \gamma_j \cdot 2P_{\mathrm{act}}\ =\ \mathrm{CPT}^{\mathrm{hom}}_{j},\end{aligned} \tag{19}$$

because $\min(\alpha_1, \alpha_2) \le \alpha_j$ and $\min(\gamma_1, \gamma_2) \le \gamma_j$ for all j. Since this holds for every j, it holds for the minimum:

$$\mathrm{CPT}^{\mathrm{het}}_{\star} \le \min_j \mathrm{CPT}^{\mathrm{hom}}_j = \mathrm{CPT}^{\mathrm{hom}}_{\star}.$$

Therefore $G_{\mathrm{het}}(s) \ge 1$ for every $s \in \mathcal{I}$, establishing claim 1.

For claim 2: if GPU type j dominates type k on both metrics ($\alpha_j \le \alpha_k$ and $\gamma_j \le \gamma_k$), then $\min(\alpha_1, \alpha_2) = \alpha_j$ and $\min(\gamma_1, \gamma_2) = \gamma_j$. Hence $\mathrm{CPT}^{\mathrm{het}}_{\star} = \mathrm{CPT}^{\mathrm{hom}}_j = \mathrm{CPT}^{\mathrm{hom}}_{\star}$ and $G_{\mathrm{het}}(s) = 1$ for every $s \in \mathcal{I}$.

Step 2: Unimodal structure (Claim 3). When neither type dominates, the metric orderings must be crossed. Under the theorem's labeling $\eta \ge 1$, the only possible crossed ordering is $\alpha_1 < \alpha_2$ and $\gamma_2 < \gamma_1$; the reverse ordering would imply $\eta < 1$. Define $r = \alpha_2/\alpha_1 > 1$ and $q = \gamma_1/\gamma_2 > 1$. The cost expressions become:

$$\mathrm{CPT}^{\mathrm{hom}}_1 = \alpha_1 \cdot m_{\mathrm{kv}}(s) + q\gamma_2 \cdot 2P_{\mathrm{act}}, \tag{20}$$

$$\mathrm{CPT}^{\mathrm{hom}}_2 = r\alpha_1 \cdot m_{\mathrm{kv}}(s) + \gamma_2 \cdot 2P_{\mathrm{act}}, \tag{21}$$

$$\mathrm{CPT}^{\mathrm{het}}_{\star} = \alpha_1 \cdot m_{\mathrm{kv}}(s) + \gamma_2 \cdot 2P_{\mathrm{act}}. \tag{22}$$

Step 2a: Crossing point. The two homogeneous CPTs are equal when

$$\alpha_1 \cdot m_{\mathrm{kv}}(s) + q\gamma_2 \cdot 2P_{\mathrm{act}} = r\alpha_1 \cdot m_{\mathrm{kv}}(s) + \gamma_2 \cdot 2P_{\mathrm{act}},$$

which gives the crossing traffic

$$x^* = \frac{(q-1)\,\gamma_2 \cdot 2P_{\mathrm{act}}}{(r-1)\,\alpha_1} = \frac{2P_{\mathrm{act}}(\gamma_1 - \gamma_2)}{\alpha_2 - \alpha_1}. \tag{23}$$

For $x < x^*$, GPU type 2 is the better homogeneous choice ($\mathrm{CPT}^{\mathrm{hom}}_2 < \mathrm{CPT}^{\mathrm{hom}}_1$). For $x > x^*$, GPU type 1 is the better homogeneous choice.

Step 2b: Monotonicity in each sub-region. Let $x = m_{\mathrm{kv}}(s)$ for brevity.

Sub-region 1 ($x \le x^*$, short sequences, homogeneous uses GPU type 2):

$$G_{\mathrm{het}}(x) = \frac{r\alpha_1 x + \gamma_2 \cdot 2P_{\mathrm{act}}}{\alpha_1 x + \gamma_2 \cdot 2P_{\mathrm{act}}}. \tag{24}$$

Differentiating,

$$\frac{dG_{\mathrm{het}}}{dx} = \frac{(r-1)\,\alpha_1 \cdot \gamma_2 \cdot 2P_{\mathrm{act}}}{(\alpha_1 x + \gamma_2 \cdot 2P_{\mathrm{act}})^2} > 0. \tag{25}$$

So G_{het} is strictly increasing in this sub-region. At $x = 0$: $G_{\mathrm{het}}(0) = (\gamma_2 \cdot 2P_{\mathrm{act}})/(\gamma_2 \cdot 2P_{\mathrm{act}}) = 1$.

Sub-region 2 ($x \ge x^*$, long sequences, homogeneous uses GPU type 1):

$$G_{\mathrm{het}}(x) = \frac{\alpha_1 x + q\gamma_2 \cdot 2P_{\mathrm{act}}}{\alpha_1 x + \gamma_2 \cdot 2P_{\mathrm{act}}}. \tag{26}$$

Differentiating,

$$\frac{dG_{\mathrm{het}}}{dx} = \frac{-(q-1)\,\alpha_1 \cdot \gamma_2 \cdot 2P_{\mathrm{act}}}{(\alpha_1 x + \gamma_2 \cdot 2P_{\mathrm{act}})^2} < 0. \tag{27}$$

So G_{het} is strictly decreasing in this sub-region. As $x \to \infty$: $G_{\mathrm{het}} \to \alpha_1/\alpha_1 = 1$.

Combining both sub-regions, the analytic gain $G_{\mathrm{het}}(x)$ increases from 1 at $x = 0$, peaks at $x = x^*$, and decreases toward 1 as $x \to \infty$. Because $m_{\mathrm{kv}}(s)$ is continuous and strictly increasing on $\mathcal{I}$, if $x^* \in m_{\mathrm{kv}}(\mathcal{I})$, there is a unique $s^* \in \mathcal{I}$ satisfying $m_{\mathrm{kv}}(s^*) = x^*$ and the restricted gain peaks there. If x^* lies outside $m_{\mathrm{kv}}(\mathcal{I})$, the feasible interval contains only one side of the analytic curve, so the restricted gain is monotone. This establishes claim 3.

Step 3: Upper-bound part of Claim 3. By the unimodality established in Step 2, the supremum of the analytic gain over $x \ge 0$ is attained at the crossing point $x = x^*$. The same upper bound therefore applies to any restricted feasible interval.

Step 3a: Peak gain at x^*. Using the sub-region 1 expression at $x = x^*$ and noting from Eq. 23 that $\alpha_1 x^* = (q-1)\gamma_2 \cdot 2P_{\mathrm{act}}/(r-1)$:

$$\begin{aligned} G_{\mathrm{het}}(x^*) &= \frac{r\alpha_1 x^* + \gamma_2 \cdot 2P_{\mathrm{act}}}{\alpha_1 x^* + \gamma_2 \cdot 2P_{\mathrm{act}}} = \frac{r \cdot \frac{(q-1)}{(r-1)} + 1}{\frac{(q-1)}{(r-1)} + 1} \\ &= \frac{r(q-1) + (r-1)}{(q-1) + (r-1)} = \frac{rq - 1}{r + q - 2}. \end{aligned} \tag{28}$$

Step 3b: Factor out cost ratio. Let $w = c_1/c_2$. Then $r = \alpha_2/\alpha_1 = \beta_1/(w\,\beta_2)$ and $q = \gamma_1/\gamma_2 = w\,F_2/F_1$. The product is cost-independent:

$$rq = \frac{\beta_1\,F_2}{\beta_2\,F_1} = \eta. \tag{29}$$

The peak gain $G_{\mathrm{het}}(x^*) = (\eta - 1)/(r + q - 2)$ is decreasing in $r + q$ for fixed $rq = \eta > 1$. To obtain the tightest bound, we minimize $r + q$ subject to $rq = \eta$:

$$r + q = \frac{\beta_1}{w\,\beta_2} + \frac{w\,F_2}{F_1}.$$

By the AM-GM inequality, $r + q \ge 2\sqrt{rq} = 2\sqrt{\eta}$, with equality when $r = q = \sqrt{\eta}$, which occurs at the cost ratio

$$w^* = \sqrt{\frac{\beta_1 F_1}{\beta_2 F_2}}.$$

Step 3c: Evaluate the bound. Substituting $r + q = 2\sqrt{\eta}$ and $rq = \eta$:

$$\begin{aligned} \sup_{x \ge 0,\, w} G_{\mathrm{het}}(x) &= \frac{\eta - 1}{2\sqrt{\eta} - 2} = \frac{(\sqrt{\eta} - 1)(\sqrt{\eta} + 1)}{2(\sqrt{\eta} - 1)} \\ &= \frac{1 + \sqrt{\eta}}{2}. \end{aligned} \tag{30}$$

This establishes the upper-bound part of claim 3 and completes the proof. □

Summary. The cost gain in the non-trivial case ($\alpha_1 < \alpha_2$, $\gamma_2 < \gamma_1$) can be expressed as

$$G_{\text{het}}(s) = \begin{cases} \dfrac{r\alpha_1 m_{\text{kv}}(s) + \gamma_2 \cdot 2P_{\text{act}}}{\alpha_1 m_{\text{kv}}(s) + \gamma_2 \cdot 2P_{\text{act}}} & \text{if } m_{\text{kv}}(s) \le x^*, \\ \dfrac{\alpha_1 m_{\text{kv}}(s) + q\gamma_2 \cdot 2P_{\text{act}}}{\alpha_1 m_{\text{kv}}(s) + \gamma_2 \cdot 2P_{\text{act}}} & \text{if } m_{\text{kv}}(s) \ge x^*. \end{cases} \tag{31}$$

The analytic gain is unimodal in $x = m_{\text{kv}}(s)$, with $G_{\text{het}}(0) = 1$, a peak of $(rq-1)/(r+q-2)$ at $x = x^*$, and $G_{\text{het}}(x) \to 1$ as $x \to \infty$. On the feasible interval $\mathcal{I}$, this peak is attained only when $x^* \in m_{\text{kv}}(\mathcal{I})$; otherwise, the restricted gain is monotone. The tightest upper bound over all non-negative traffic values and all cost ratios is

$$\sup_{x \ge 0,\, w} G_{\text{het}}(x) = \frac{1 + \sqrt{\beta_1 F_2/(\beta_2 F_1)}}{2}.$$

A.6 CAD Feasibility Condition for Uniform-Layer Models

This section gives a simple condition for the feasibility mismatch discussed in Section 3.4. The condition applies to the uniform-layer CAD setting, where every layer has the same attention pattern and therefore the attention side can be described by a single feasible latency range. Hybrid-attention architectures, where full-attention, sliding-window-attention, or linear-attention layers may have different latency and memory behavior, require the separate discussion in Section 3.4.

Following the heterogeneous notation in Section 3.1, let GPU type 1 serve the attention side and GPU type 2 serve the rest-of-model side. Let M_1 be the memory capacity of one GPU of type 1, β_1 be its memory bandwidth, and μ be the number of micro-batches sharing the attention-side memory. Then a simple upper bound on the feasible attention-stage latency is

$$T_{\text{attn}}^{\max} = \frac{M_1}{\mu\beta_1}. \tag{32}$$

This expression reflects that each micro-batch can use at most M_1/μ bytes of attention-side memory, which can be streamed at bandwidth β_1.

Let π_2 denote the parallelism strategy on the rest-of-model side. Let $M_{\text{weights},2}(\pi_2)$ be the per-GPU model-weight footprint on GPU type 2 under this strategy, and let β_2 be the memory bandwidth of GPU type 2. The rest-of-model stage must at least load its shard of the model weights, so its latency is lower-bounded by

$$T_{\text{rest}}^{\min}(\pi_2) = \frac{M_{\text{weights},2}(\pi_2)}{\beta_2}. \tag{33}$$

For example, under an even sharding strategy over $g_2(\pi_2)$ GPUs, one can approximate

$$M_{\text{weights},2}(\pi_2) \approx \frac{M_{\text{weights}}}{g_2(\pi_2)}. \tag{34}$$

For CAD to be feasible under this simplified uniform-layer model, the feasible latency ranges of the two stages must overlap. A necessary condition is therefore

$$T_{\text{attn}}^{\max} \ge T_{\text{rest}}^{\min}(\pi_2), \qquad \text{i.e.,} \qquad \frac{M_1}{\mu\beta_1} \ge \frac{M_{\text{weights},2}(\pi_2)}{\beta_2}. \tag{35}$$

When this inequality fails, even the largest feasible attention-stage latency is smaller than the unavoidable weight-loading latency on the rest-of-model side. In that case, the two CAD stages cannot be balanced, and pipeline bubbles are unavoidable.

B Detailed Serving-Plan Formulation

This appendix section expands the compact formulation in Section 4. The main text keeps only the abstractions needed to motivate the planner; here we spell out the structural definitions, batching notation, execution and communication latency, and memory constraint.

B.1 Model Representation and Partition

We model one decode step as a computation DAG $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where each node $v \in \mathcal{V}$ is an operator and each edge $(u, v) \in \mathcal{E}$ is a data dependency. During serving, operators are executed according to a fixed topological order. For transformer inference this order is effectively a linear operator sequence

$$\mathbf{o} = (o_1, o_2, \ldots, o_{N_{\text{model}}}),$$

and all partitioning decisions below refer to positions in this sequence.

Let L_{model} be the number of transformer layers. A layer boundary is the first operator of a layer; we denote these positions by $\ell_1 < \ell_2 < \cdots < \ell_{L_{\text{model}}}$, with $\ell_1 = 1$. A **partition block** is the minimal layer-aligned repeating unit of the operator sequence. Formally, it spans L_{block} consecutive layers, or equivalently N_{block} consecutive operators, where L_{block} is the smallest positive period such that the operator pattern of layers $\ell_k, \ldots, \ell_{k+L_{\text{block}}-1}$ repeats for every block-aligned starting layer k. We take the first partition block to start at ℓ_1 and assume L_{block} divides L_{model}, giving $B_{\text{total}} = L_{\text{model}}/L_{\text{block}}$ partition blocks.

Within one partition block, the optimizer chooses a stage template

$$\mathbf{q} = (q_0, q_1, \ldots, q_S), \qquad 0 = q_0 < q_1 < \cdots < q_S = N_{\text{block}}.$$

Stage j contains operators at relative positions $q_{j-1}+1, \ldots, q_j$ inside the block. The same template is applied to every partition block in the model, so stage j denotes the same relative operator slice in each repeated block.

B.2 Device-Group Types, Replicas, and Placement

A device-group type m is a logical configuration (h_m, π_m), where h_m is the GPU type and π_m is the intra-group parallelism strategy, such as tensor parallelism, expert parallelism,

or their composition. A device-group replica is one physical instance of that type. Replicas of the same type have identical configuration and execute the same stage set, but serve disjoint subsets of the global resident batch.

We restrict the number of stages to $S = kK$ for an integer $k \geq 1$. Stages are assigned to the ordered device-group types by the round-robin map

$$\sigma(j) = ((j-1) \bmod K) + 1, \qquad j = 1, \ldots, S. \tag{36}$$

This map is applied identically within every partition block. Consequently, each type owns exactly k stage positions per block, and the stage-to-type assignment repeats with the same partition template. The stage set assigned to type m is therefore

$$\mathcal{S}_m = \{j : 1 \leq j \leq S,\ \sigma(j) = m\}.$$

Every replica of type m executes all stages in $\mathcal{S}_m$.

Let B be the global resident batch size across the full deployment. If a replica of type m has resident batch size b_m, then the number of replicas of that type is

$$n_m = \frac{B}{b_m}, \qquad m = 1, \ldots, K, \tag{37}$$

which requires b_m to divide B. This coupling ensures that all device-group types collectively process the same global batch.

B.3 Scheduling Parameters

The resident batch on each replica is split into μ microbatches. We write

$$\widehat{B} = \frac{B}{\mu}, \qquad \widehat{b}_m = \frac{b_m}{\mu},$$

for the global microbatch size and per-replica microbatch size, respectively. The divisibility requirements are $\mu \mid B$ and $\mu \mid b_m$ for all device-group types. The resident batch b_m determines the number of tokens completed by a replica per round trip, while $\widehat{b}_m$ determines the work executed in one pipeline scheduling step.

The execution latency of stage j is

$$t_j = f_{\text{lat}}\Big(\mathbf{o}_{q_{j-1}+1:q_j}, (h_{\sigma(j)}, \pi_{\sigma(j)}), \widehat{b}_{\sigma(j)}\Big), \qquad j = 1, \ldots, S. \tag{38}$$

Let A_j denote the activation interface at the boundary after stage j, including its tensor shape, byte size per request, and parallel layout. Define $j^+ = j + 1$ for $j < S$ and $j^+ = 1$ for $j = S$; the latter is the boundary between consecutive partition blocks. The corresponding transfer latency is

$$\begin{aligned} u_j = f_{\text{comm}}\big(&A_j, h_{\sigma(j)}, \pi_{\sigma(j)}, \widehat{b}_{\sigma(j)}, \\ &h_{\sigma(j^+)}, \pi_{\sigma(j^+)}, \widehat{b}_{\sigma(j^+)}\big), \end{aligned} \tag{39}$$

where f_{comm} is obtained from communication profiles and accounts for any resharding or redistribution between the source and destination replicas. It returns the local handoff cost when the adjacent stages execute on the same replica. The $j = S$ transfer occurs between consecutive blocks and is omitted after the final block.

The steady-state round-trip time is then

$$T_{\text{rtt}} = f_{\text{rtt}}(\mu, B_{\text{total}}, \mathbf{t}, \mathbf{u}), \tag{40}$$

where $\mathbf{t} = (t_1, \ldots, t_S)$ and $\mathbf{u} = (u_1, \ldots, u_S)$. The scheduling model enforces execution and transfer dependencies, models contention for compute and communication resources, and permits overlap only when the selected schedule and resources allow it. The planner evaluates this function by lightweight simulation.

B.4 Memory Constraint

For each device-group type m, $M_{\text{weights},m}$ denotes the weight memory for the stages assigned to that type, including all repeated partition blocks hosted by a replica. Let $\text{KV}_m(\widehat{b}_m, s)$ denote the KV-cache memory required by one per-replica microbatch at sequence length s; this term is zero for stages that do not own KV-resident attention state. Since all μ microbatches are resident in the pipeline, a replica of type m must satisfy

$$M_{\text{weights},m} + \mu\, \text{KV}_m(\widehat{b}_m, s) \leq M_m,$$

where M_m is the effective memory capacity of the device group after accounting for the parallelism strategy π_m. Together with the SLO constraint and the objective in equation 9a, this yields the compact optimization problem in equation 9. The cost-per-token objective is derived from global batch size and replica counts in Appendix B.5.

B.5 Derivation of the Cost-per-Token Objective

We derive the general serving-plan objective CPT_{plan} in equation 9a from the perspective of the global batch size and device-group replicas introduced in Section B.2.

Setup. Consider a global resident batch of B requests processed in parallel across all replicas. Each replica of device-group type m holds a per-replica resident batch of b_m requests. When the schedule uses μ micro-batches, the global micro-batch size is $\widehat{B} = B/\mu$ and the per-replica micro-batch size is $\widehat{b}_m = b_m/\mu$. The system therefore requires

$$n_m = \frac{B}{b_m} = \frac{\widehat{B}}{\widehat{b}_m} \tag{41}$$

replicas of type m.

Throughput. In steady-state pipeline execution, each replica completes one decoding step (i.e., generates one output token for each of its $b_m = \mu\widehat{b}_m$ resident requests) every T_{rtt} seconds. Since there are $n_m = B/b_m$ replicas of type m, and every request passes through every device-group type, the system-wide throughput is determined by the global resident batch:

$$\text{Throughput} = \frac{B}{T_{\text{rtt}}} \qquad \text{tokens/s}. \tag{42}$$

Total cost rate. The instantaneous monetary cost rate of operating all replicas is

$$\mathrm{CostRate}_{\mathrm{total}} \;=\; \sum_{m=1}^{K} n_m \cdot c_m \;=\; \sum_{m=1}^{K} \frac{B}{b_m}\, c_m \;=\; B \sum_{m=1}^{K} \frac{c_m}{b_m}. \tag{43}$$

Cost per token. Dividing the total cost rate by the throughput yields

$$\begin{aligned} \mathrm{CPT}_{\mathrm{plan}} \;&=\; \frac{\mathrm{CostRate}_{\mathrm{total}}}{\mathrm{Throughput}} \;=\; \frac{B \displaystyle\sum_{m=1}^{K} \frac{c_m}{b_m}}{\dfrac{B}{T_{\mathrm{rtt}}}} \\ &=\; T_{\mathrm{rtt}} \;\cdot\; \sum_{m=1}^{K} \frac{c_m}{b_m}, \end{aligned} \tag{44}$$

which recovers equation 9a. Note that the global batch size B cancels, confirming that the cost-per-token objective depends only on the device-group configurations and per-replica resident batch sizes, equivalently on $\mu\widehat{b}_m$, not on the absolute scale of the deployment. This cancellation assumes scale-invariant replication: adding replicas repeats the same per-replica execution and transfer pattern.

C Planner Details for Section 5

This appendix details the latency and cost lower bounds used by the branch-and-bound search in Section 5.3. These bounds allow the planner to prune partial assignments before invoking the schedule simulator.

C.1 Branch-and-Bound Lower Bounds

After selecting a prefix $P_i = (p_1, \ldots, p_i)$ of owner points, the planner computes an optimistic completion using the best remaining coordinate from each unassigned frontier.

For a point p of owner m, the mandatory serialized compute work is $C_m(p) = \mu \sum_{j \in \mathcal{S}_m} t_j(p)$. The compute lower bound for a prefix is therefore

$$L_{\mathrm{comp}}(P_i) = \max\left\{ \max_{m \le i} C_m(p_m), \max_{m > i} \min_{p \in \mathcal{F}_m} C_m(p) \right\}. \tag{45}$$

The planner similarly lower-bounds communication by accumulating mandatory serialized transfer work at each network endpoint. A boundary whose endpoint has not yet been assigned is minimized over that owner's feasible replica counts. Boundaries are minimized independently, which can only make the estimate optimistic. Denoting this endpoint bound by $L_{\mathrm{net}}(P_i)$ gives the valid latency lower bound

$$L_T(P_i) = \max\left\{ L_{\mathrm{comp}}(P_i), L_{\mathrm{net}}(P_i) \right\} \le T_{\mathrm{rtt}}. \tag{46}$$

Suffix minima also lower-bound the normalized cost of any completion; let $R_{\mathrm{LB}}(P_i)$ denote this bound. The best possible objective below the prefix satisfies $\mathrm{CPT}_{\mathrm{LB}}(P_i) = L_T(P_i)\, R_{\mathrm{LB}}(P_i) \le \mathrm{CPT}_{\mathrm{plan}}$.

D Notation

Symbol	Description
Model and workload	
$\mathbf{o}$	Operator sequence $(o_1, \ldots, o_{N_{\text{model}}})$ of one decode step
N_{model}	Total number of operators
L_{model}	Total number of transformer layers
P_{act}	GEMM-side parameters activated per request after model-parallel sharding (routed experts only for MoE)
M_{weights}	Resident model-weight footprint after sharding, including inactive experts
$D^{\text{attn}}(b)$, $D^{\text{gemm}}(b)$	Aggregated attention and GEMM memory traffic at batch size b
$W^{\text{gemm}}(b)$	Aggregated GEMM FLOPs at batch size b
s	Average sequence length (workload parameter)
$m_{\text{kv}}(s)$	Per-request KV-cache memory at sequence length s
b^*	GEMM compute-bound threshold batch size
$b_{\max}(s)$	Max feasible batch size under memory at sequence length s
$\bar{b}(s)$	Continuous relaxation of $b_{\max}(s)$ used in closed-form analysis
GPU hardware (per GPU type j)	
c_j	Unit-time monetary cost of GPU type j (\$/s); shorthand c in the single-type case, c_1, c_2 for two types
β_j	Memory bandwidth (bytes/s); shorthand β, or β_1, β_2
F_j	Peak compute throughput (FLOPS); shorthand F, or F_1, F_2
M_j	Memory capacity (bytes); shorthand M, or M_1, M_2
α_j, γ_j	Monetary cost per byte of memory traffic and per FLOP, respectively, for GPU type j
CPT	Cost per token; CPT_{plan} is the serving-plan objective; superscripts coloc/hom/het denote serving modes
Theoretical analysis (Section 3)	
$G_{\text{hom}}(s)$	Cost gain of homogeneous ODS over colocated serving
$G_{\text{het}}(s)$	Cost gain of heterogeneous over homogeneous ODS
η	Hardware ratio $(\beta_1 F_2)/(\beta_2 F_1)$ of two GPU types
Model structure (partition block; Section 4)	
L_{block}	Number of transformer layers per partition block
N_{block}	Number of operators per partition block
B_{total}	Number of partition blocks ($= L_{\text{model}}/L_{\text{block}}$)
Decision variables (Section 4)	
S	Number of stages per partition block
$\mathbf{q}$	Stage-boundary offsets $(q_0, \ldots, q_S)$ within a block
K	Number of device-group types
(h_m, π_m)	Device-group configuration (GPU type, parallelism) of type m
b_m	Per-replica resident batch size of device-group type m
$\widehat{b}_m$	Per-replica micro-batch size of device-group type m ($= b_m/\mu$)
μ	Number of micro-batches
Derived / scheduling quantities (Section 4)	
$\mathcal{S}_m$	Stages assigned to device-group type m (placement group)
$\sigma(j)$	Placement map from stage j to its device-group type
B	Global resident batch size across all replicas
n_m	Replicas of device-group type m ($= B/b_m$)
c_m	Unit-time monetary cost of one replica of device-group type m
M_m	Effective memory capacity of a replica of type m under π_m
$M_{\text{weights},m}$	Total weight memory of stages assigned to type m
$\text{KV}_m(\widehat{b}_m, s)$	KV-cache memory for one per-replica micro-batch on a replica of type m
t_j	Execution latency of stage j
u_j	Activation-transfer latency at the boundary after stage j
T_{rtt}	Pipeline round-trip time (per-token latency in steady state)
T_{SLO}	TPOT service-level objective
Planner (Section 5)	
L_{sub}, n	Sub-block length in layers and repetitions per block ($n = L_{\text{block}}/L_{\text{sub}}$)
$\mathbf{q}_{\text{sub}}$	Sub-block stage template, tiled over the n repetitions
$\mathbf{T}_m(p)$	Stage-latency vector of owner m's point p over $\mathcal{S}_m$
$r_m(p)$, $\rho_m(p)$	Normalized cost and GPU count of point p: $c_m(p)/b_m(p)$, $g_m(p)/b_m(p)$
$\mathcal{F}_m$	Network-safe Pareto frontier of owner m

Table 4. Unified notation used throughout the paper.