IHS-LM: Intra-batch Hybrid Scheduling and Layer Migration for VLM Pipeline Inference Acceleration on Edge Devices

Yun Li
College of Computer Science and Software Engineering, Hohai University
Nanjing, China
260607040001@hhu.edu.cn

Tianfu Pang
College of Computer Science and Software Engineering, Hohai University
Nanjing, China
250207010004@hhu.edu.cn

Zhiyu Cai
College of Computer Science and Software Engineering, Hohai University
Nanjing, China
260407010001@hhu.edu.cn

Zhenxiang Pan
College of Computer Science and Software Engineering, Hohai University
Nanjing, China
240407010002@hhu.edu.cn

Yingchi Mao*
College of Computer Science and Software Engineering, Hohai University
Nanjing, China
yingchimao@hhu.edu.cn

Jie Wu
Cloud Computing Research Institute, China Telecom
Beijing, China
Temple University
Department of Computer and Information Sciences
Philadelphia, PA, USA
jiewu@temple.edu

Abstract

Vision-Language Models (VLMs) are increasingly deployed via distributed inference to perform visual perception and semantic reasoning tasks on edge devices. Pipeline parallelism is well-suited for collaborative edge inference, as it partitions models across devices with minimal communication between adjacent stages. However, pipeline-parallel inference on edge devices faces two challenges. First, prior intra-batch scheduling methods determine the prefill–decode token ratio without considering available KV cache capacity and token distribution, resulting in intra-batch workload imbalance. Second, static model partitioning methods assign layers to devices without accounting for dynamic bandwidth variations, leading to load imbalance across pipeline stages. These imbalances result in pipeline bubbles and increased inference latency. To address these challenges, we present IHS-LM, a low-latency pipeline-parallel inference framework for VLM serving on heterogeneous edge devices. Specifically, to alleviate intra-batch workload imbalance, IHS-LM introduces an intra-batch Hybrid Scheduling method (IHS), which dynamically adjusts the prefill–decode token ratio based on token distribution and available KV cache capacity. To mitigate load imbalance across stages, IHS-LM proposes a Bandwidth-Aware Model Layer Migration method (BAMLM), which detects imbalance caused by dynamic bandwidth and resolves imbalance through fine-grained layer migration. We compare IHS-LM with existing pipeline-parallel inference methods in high-load and heterogeneous bandwidth settings. Experiments show that IHS-LM achieves the lowest inference latency, while preserving model accuracy.

CCS Concepts

• **Computer systems organization** → **Distributed architectures**; **Parallel architectures**; • **Computing methodologies** → *Parallel computing methodologies.*

Keywords

Vision-Language Models; Pipeline Parallelism; Edge Inference; Intra-batch Scheduling; Layer Migration

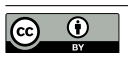
ACM Reference Format:
Yun Li, Tianfu Pang, Zhiyu Cai, Zhenxiang Pan, Yingchi Mao, and Jie Wu. 2026. IHS-LM: Intra-batch Hybrid Scheduling and Layer Migration for VLM Pipeline Inference Acceleration on Edge Devices. In *Proceedings of the 55th International Conference on Parallel Processing (ICPP '26), September 28–October 1, 2026, Singapore.* ACM, New York, NY, USA, 11 pages.

1 Introduction

Vision-Language Models (VLMs) have demonstrated remarkable capabilities [10, 23, 31] in visual understanding and multimodal reasoning, enabling diverse applications in edge computing [13, 22, 34], such as inspection of industrial defects and monitoring of infrastructure. However, the deployment of VLMs on edge devices is severely hindered by two primary bottlenecks. First, the large size of VLMs often exceeds the memory capacity of individual edge devices. Second, offloading inference to the cloud incurs prohibitive latency and raises privacy concerns. To overcome these bottlenecks, collaborative inference across multiple edge devices has emerged as a promising solution.

Collaborative inference [3, 20] relies on distributed parallelism strategies. Prior studies [15, 24] indicate that data parallelism (DP) and sequence parallelism (SP) require full model replication on each

*Corresponding author.



ICPP '26, Singapore

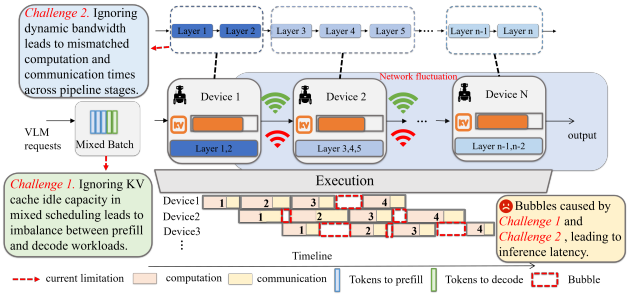


Figure 1: Two Challenges in Pipeline-Parallel Inference on Dynamic Edge Devices.

device, exceeding the memory capacity of resource-constrained edge devices. Tensor parallelism (TP) [2, 18] incurs prohibitive communication overhead due to frequent all-reduce operations at each layer. In contrast, pipeline parallelism (PP) [8, 17, 25] partitions the model into sequential stages and communicates activations between adjacent stages. PP is well suited for bandwidth-constrained and memory-limited edge devices due to its low communication overhead and balanced memory distribution [7]. However, as illustrated in Figure 1, sustaining efficient PP inference in dynamic edge devices faces two challenges.

Challenge 1. The inference process of VLMs comprises a compute-intensive prefill phase and a memory-bandwidth-bound decode phase [4, 36]. Due to the distinct resource requirements of the prefill and decode phases, intra-batch scheduling methods must carefully manage the prefill–decode token ratio within each batch, taking into account the token distribution of pending requests [14, 35]. However, as illustrated in Figure 1, prior scheduling methods fail to balance prefill tokens from newly arrived requests and decode tokens from ongoing requests, leading to prefill–decode imbalance and resource underutilization. Specifically, FasterTransformer [19] adopts non-preemptive batch-level scheduling, requiring all requests in a batch to complete before admitting new ones. Orca [29] enables iteration-level dynamic batching to admit new requests, but long prefill requests can still block ongoing decode requests. Sarathi-Serve [1] mitigates prefill-induced stalls via chunked prefilling and hybrid scheduling, yet cannot dynamically adjust the prefill–decode ratio based on available KV cache capacity and token distribution. These limitations lead to imbalance between prefill and decode workloads, thereby increasing inference latency.

Challenge 2. In pipeline-parallel edge inference, the balance of stage latency (computation and communication) is critical to system performance. When link bandwidth fluctuates, latency disparities across stages can increase significantly, disrupting the load balance across pipeline stages established by static model partitioning. Dynamic layer migration restores stage balance by reassigning model layers across devices at runtime [9, 33]. However, as illustrated in Figure 1, existing approaches overlook bandwidth heterogeneity, leading to inter-stage imbalance that remains unresolved even after migration. Specifically, DynaPipe [9] ignores bandwidth heterogeneity and limits migration to the pipeline tail. LinguaLinked [33] cannot detect inter-stage imbalance during inference and adapt to dynamic bandwidth due to reliance on offline partitioning.

In this paper, we address these challenges by introducing IHS-LM, a low-latency pipeline-parallel inference framework for VLM serving on heterogeneous edge devices, guided by the following design goals: (1) Alleviate intra-batch workload imbalance to enable high-throughput inference for heterogeneous requests. (2) Mitigate load imbalance across pipeline stages to reduce idle time and minimize latency.

To achieve these design goals, IHS-LM introduces two key modules: the intra-batch Hybrid Scheduling (IHS) module and the Bandwidth-Aware Model Layer Migration (BAMLM) module. The IHS dynamically regulates the prefill–decode token ratio within each global batch by jointly monitoring pending-request token counts and available KV cache capacity, thereby alleviating intra-batch load imbalance and reducing inference latency. The BAMLM module is composed of two tightly integrated components. The first is a Sliding Window Bandwidth Sensing mechanism, which monitors per-stage latency and smooths fluctuations to reliably detect pipeline imbalance. The second is a Fine-Grained Model Layer Migration strategy, which pre-partitions candidate boundary layers offline and dynamically selects the optimal layers to migrate online based on real-time stage latency differences. These components enable adaptive, low-latency load balancing across all pipeline stages, ensuring efficient utilization of heterogeneous edge resources.

In summary, this paper makes the following contributions.

- An intra-batch hybrid scheduling mechanism that dynamically adjusts the prefill–decode token ratio based on token distribution and available KV cache capacity, achieving balanced prefill and decode workloads.
- A bandwidth-aware dynamic layer migration mechanism that mitigates load imbalance caused by bandwidth variations by migrating model layers across pipeline stages.
- We implement and evaluate IHS-LM on a real heterogeneous edge cluster comprising multiple NVIDIA Jetson devices. Compared with baselines, IHS-LM achieves the lowest inference latency while preserving model accuracy.

2 System Overview

Figure 2 illustrates an overview of IHS-LM with three workers for clarity; the system generalizes to N heterogeneous worker nodes. IHS-LM adopts a two-tier coordinator-worker architecture to address both intra-batch and inter-stage load imbalance. The coordinator serves as the central control plane, responsible for global scheduling, imbalance detection, and migration orchestration. The N worker nodes collectively form the distributed inference pipeline, each executing assigned model layer computations and monitoring local stage latency.

Upon receiving multimodal requests, the coordinator invokes the intra-batch Hybrid Scheduling (IHS) module, which performs token throttling to construct a mixed batch by dynamically balancing prefill and decode tokens based on pending token counts and available KV cache capacity (①). The mixed batch is then dispatched into the worker pipeline (②), where each worker executes forward computation over its assigned model layers and forwards intermediate activations to the downstream worker. Concurrently, each worker's Local Monitor records per-batch stage latency and feeds it into the

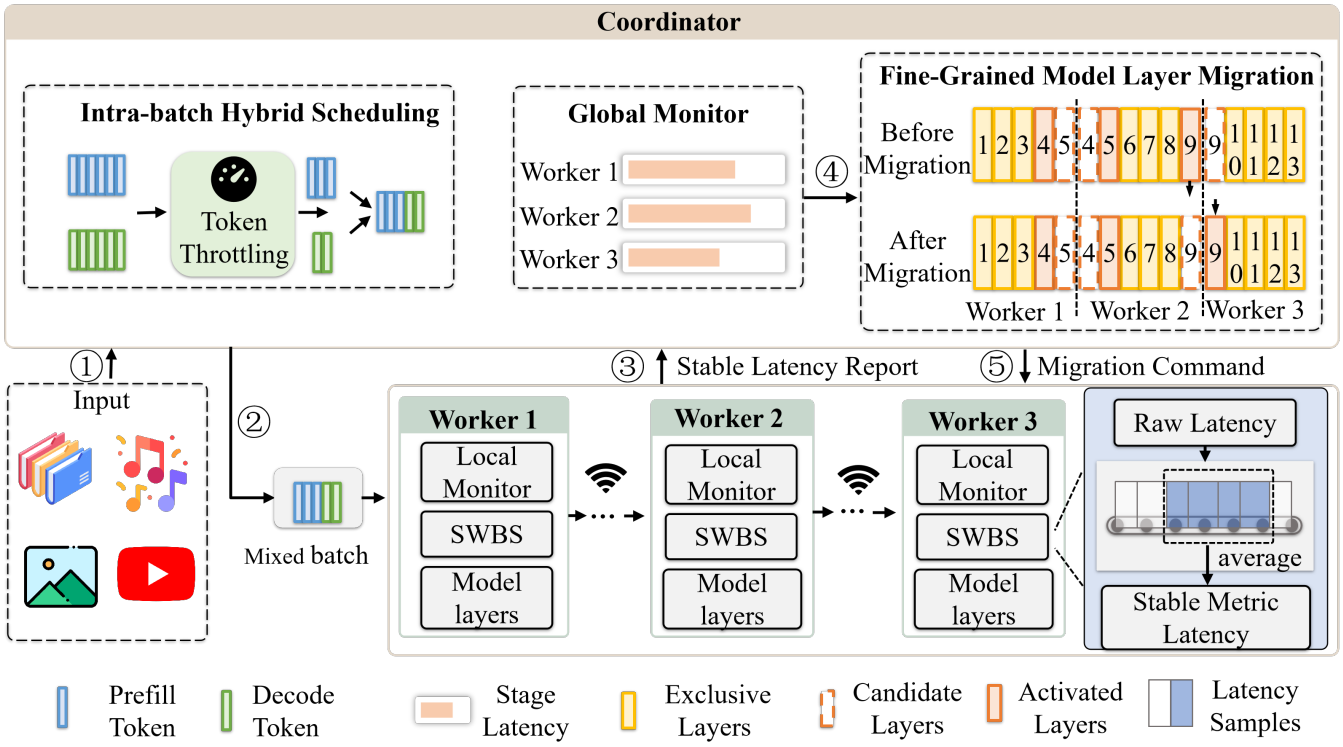


Figure 2: Overview of IHS-LM.

Sliding Window Bandwidth Sensing (SWBS) module to filter transient noise, producing a stable latency estimate that is periodically uploaded to the coordinator (③). The coordinator's Global Monitor aggregates these reports and detects persistent imbalance when the stage latency range exceeds the dynamic threshold (④). Upon detection, the Migration Decision Engine computes the optimal layer reassignment across all stage boundaries and broadcasts migration commands to the affected workers, where candidate layers pre-loaded on adjacent devices are activated or frozen via logical operator routing—shifting the computational boundary without any physical weight transmission (⑤).

The two core modules, IHS and BAMLM, operate at complementary timescales. IHS functions at the per-batch level to mitigate intra-batch workload skew, whereas BAMLM works at a coarser granularity to correct persistent inter-stage imbalance resulting from bandwidth fluctuations. Together, they form a closed-loop adaptation system that continuously reduces pipeline bubbles in dynamic edge environments.

3 Intra-Batch Hybrid Scheduling for Edge VLM Inference Acceleration

During edge collaborative inference, dynamic request arrivals and the divergent computational characteristics of the prefill and decode phases introduce intra-batch workload skew and uneven micro-batch execution times. Specifically, VLM inference exhibits significant phase-level disparities: the prefill phase performs compute-intensive image encoding and prompt processing, whereas the decode phase executes memory-bandwidth-intensive autoregressive token generation. This divergence renders batch-level or FIFO scheduling inadequate, causing pipeline bubbles from GPU memory stalls or computational synchronization delays, thereby increasing end-to-end latency.

To address this, we propose an intra-batch Hybrid Scheduling (IHS) method that introduces a fine-grained token flow control mechanism. At each iteration, IHS dynamically regulates the prefill–decode token ratio within the global batch by monitoring the number of pending prefill/decode tokens and the KV cache free ratio on edge devices. The method comprises three successive stages: prefill quota computation, decode quota computation, and hybrid batch construction.

3.1 Prefill Quota Computation

The prefill quota B_{pre} is computed based on the number of tokens pending prefill and the KV cache free ratio. At timestamp t, in a pipelined cluster of N heterogeneous edge devices, the global KV cache free ratio KV_{free} is defined as the minimum memory ratio across all devices:

$$KV_{free} = \min_{i=1}^{N} \left(KV_{free}^{(i)} \right). \tag{1}$$

Let W_{pre} denote the total number of tokens pending prefill in Q_{wait}. The prefill quota derives from two perspectives.

From the load perspective, to consume all pending tokens within K_{step} iterations, the load-driven quota $B_{pre}^{(load)}$ clips the baseline demand W_{pre}/K_{step} to the hardware-permissible interval $[B_{\min}, B_{\max}]$:

$$B_{pre}^{(load)} = \min\left(\max\left(\frac{W_{pre}}{K_{step}},\ B_{min}\right),\ B_{max}\right). \tag{2}$$

From the memory perspective, the resource-driven quota $B_{pre}^{(res)}$ scales the batch size linearly with KV_{free}, ensuring maximum throughput under sufficient memory while throttling under scarcity:

$$B_{pre}^{(res)} = \max\left(B_{max} \times KV_{free},\ B_{min}\right). \tag{3}$$

Since relying solely on $B_{pre}^{(load)}$ risks memory overflow and $B_{pre}^{(res)}$ alone lacks safety margin for decoding, the system introduces a memory safety waterline KV_{thresh}—the minimum reserved memory ratio for continuous KV cache growth during decoding. Let $[x]^+ = \max(x, 0)$ and $\eta = \left[(KV_{free} - KV_{thresh})/(1 - KV_{thresh})\right]^+$. Integrating backlog demand, resource constraints, and this safety waterline, the initial prefill quota is:

$$B_{pre}^{init} = \min\left\{W_{pre}, \max\left(\frac{W_{pre}}{K_{step}}, B_{min}\right), B_{max}\eta\right\}. \tag{4}$$

When $KV_{free} < KV_{thresh}$, η becomes zero, and the scheduler temporarily stops admitting new prefill tokens to reserve GPU memory headroom for ongoing decoding. When sufficient KV cache space is available, B_{min} serves as the minimum effective scheduling granularity, while the quota is still clipped by the number of pending prefill tokens.

3.2 Decode Quota Computation

Unlike prefill, which completes within a single iteration, decoding spans multiple steps to generate tokens autoregressively. Consequently, the running queue $\mathcal{Q}_{run}$ exhibits high stability: its composition changes only when new requests join or existing ones complete, with the majority of requests carried over from the previous iteration.

Leveraging this stability, the system employs a pipeline-based load averaging strategy rather than complex dynamic regulation. The scheduler first counts the total number of active decoding tokens $W_{dec} = \text{TokenCount}(Q_{run})$. To respect the per-iteration token budget, it admits at most $\widetilde{W}_{dec} = \min(W_{dec}, B_{max})$ decode tokens in the current iteration and distributes them evenly across pipeline depth D_{pp}, yielding the per-micro-batch decode quota:

$$B_{dec}^{(quota)} = \lceil \widetilde{W}_{dec}/D_{pp} \rceil \tag{5}$$

To handle tail remainders when $\widetilde{W}_{dec}$ is not evenly divisible, the actual number of tokens scheduled per micro-batch is

$$B_{dec}^{(i)} = \min\left(\widetilde{W}_{dec}^{rem,i}, B_{dec}^{quota}\right), \tag{6}$$

where $\widetilde{W}_{dec}^{rem,i}$ denotes the number of remaining scheduled decode tokens before constructing the i-th micro-batch. This ensures all tokens are scheduled when the total falls below the quota, and caps at the quota otherwise. By forcing each micro-batch to carry an equal token load (except the tail), the method eliminates inter-micro-batch imbalance and prevents oversized individual batches from stalling the pipeline rhythm.

Algorithm 1 Intra-batch Hybrid Scheduling Algorithm (IHS)

Require: Waiting queue Q_{wait}, running queue Q_{run}, KV cache free ratio KV_{free}, pipeline depth D_{pp}, capacity limits B_{min}, B_{max}, memory threshold KV_{thresh}, target steps K_{step}

Ensure: Micro-batch set $\{T_{micro}^{(1)}, \ldots, T_{micro}^{(D_{pp})}\}$

$KV_{free} \leftarrow \min_i(KV_{free}^{(i)})$
$W_{pre} \leftarrow \text{TokenCount}(Q_{wait})$
$W_{dec} \leftarrow \text{TokenCount}(Q_{run})$
$\widetilde{W}_{dec} \leftarrow \min(W_{dec},\ B_{max})$
$\eta \leftarrow \max\left(\frac{KV_{free} - KV_{thresh}}{1 - KV_{thresh}},\ 0\right)$
$B_{pre}^{init} \leftarrow \min\left(W_{pre},\ \max(\frac{W_{pre}}{K_{step}},\ B_{min}),\ B_{max}\eta\right)$
$B_{pre} \leftarrow \min\left(B_{pre}^{init},\ \max(B_{max} - \widetilde{W}_{dec},\ 0)\right)$
$T_{pre} \leftarrow \text{Fetch}(Q_{wait}, B_{pre})$
$T_{\text{dec\_all}} \leftarrow \text{Fetch}(Q_{run}, \widetilde{W}_{dec})$
for $i = 1$ to D_{pp} **do**
  $n_{dec}^{(i)} \leftarrow \left\lceil \frac{\widetilde{W}_{dec}}{D_{pp}} \right\rceil$
  $T_{dec}^{(i)} \leftarrow \text{Pop}\left(T_{\text{dec\_all}},\ \min\left(\text{TokenCount}(T_{\text{dec\_all}}),\ n_{dec}^{(i)}\right)\right)$
  $S_{slice} \leftarrow \left\lfloor \frac{B_{pre}}{D_{pp}} \right\rfloor$
  if $i = D_{pp}$ **then**
    $S_{slice} \leftarrow \text{TokenCount}(T_{pre})$
  end if
  $T_{micro}^{(i)} \leftarrow \text{Concat}\left(T_{dec}^{(i)},\ \text{Pop}(T_{pre}, S_{slice})\right)$
end for
return $\{T_{micro}^{(1)}, \ldots, T_{micro}^{(D_{pp})}\}$

3.3 Hybrid Batch Construction

After determining the decode quota, the system constructs the global batch following a decode-first principle. Although B_{pre}^{init} accounts for pending prefill tokens and available KV cache capacity, its effective capacity is further constrained by the hardware per-iteration token limit B_{max} minus the budget already consumed by the scheduled decode tokens. The final admitted prefill quota must therefore satisfy:

$$B_{pre} = \min\left(B_{pre}^{init},\ \max(B_{max} - \widetilde{W}_{dec},\ 0)\right). \tag{7}$$

To convert the scheduling plan into hardware execution instructions, the scheduler fetches B_{pre} tokens from $\mathcal{Q}_{wait}$ to form the prefill token set $\mathcal{T}_{pre}$, and partitions the active tokens in $\mathcal{Q}_{run}$ into D_{pp} micro-batches $\{\mathcal{T}_{dec}^{(1)}, \ldots, \mathcal{T}_{dec}^{(D_{pp})}\}$. To prevent prefill concentration from causing pipeline stalls, $\mathcal{T}_{pre}$ is evenly distributed across micro-batches (with any remainder assigned to the last). Each micro-batch is then constructed via logical concatenation:

$$\mathcal{T}_{micro}^{(i)} = \text{Concat}\left(\mathcal{T}_{dec}^{(i)},\ \text{Slice}\left(\mathcal{T}_{pre},\ \text{size} \approx \frac{B_{pre}}{D_{pp}}\right)\right). \tag{8}$$

This interleaving ensures computational consistency across micro-batches, enabling fine-grained scheduling within the same pipeline

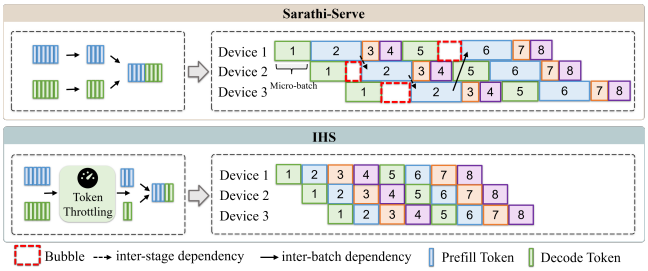


Figure 3: Comparison of Pipeline Execution Timing and Pipeline Bubbles between Sarathi-Serve [1] and IHS.

stage to effectively reduce pipeline bubbles and end-to-end inference latency. The complete scheduling procedure is detailed in Algorithm 1.

Figure 3 compares pipeline bubble distributions between Sarathi-Serve and the proposed IHS method. Although Sarathi-Serve employs chunked prefilling, its lack of global GPU memory and token-count awareness leads to unbalanced computation across micro-batches and noticeable pipeline stalls. In contrast, IHS dynamically distributes prefill and decode tokens evenly across micro-batches, achieving consistent execution times and thereby reducing pipeline bubbles and inference latency.

4 Bandwidth-Aware Layer Migration for Edge VLM Inference Acceleration

Rather than relying on noisy instantaneous bandwidth measurements, SWBS uses communication-aware stage latency as a proxy signal, because bandwidth degradation is ultimately reflected in the communication component of pipeline stage latency. We propose BAMLM, a bandwidth-aware dynamic layer migration method for distributed pipeline inference. BAMLM integrates two components: a Sliding Window Bandwidth Sensing (SWBS) mechanism that smooths network fluctuations to reliably trigger migration, and a Fine-Grained Model Layer Migration (FGMLM) strategy that pre-partitions candidate boundary layers offline and dynamically selects the optimal migration set online, achieving stage-level load balance without physical weight transmission.

4.1 Problem Formulation

In pipeline-parallel inference, each device d_i hosts a contiguous layer subset $\mathcal{L}^{(t)}_{\text{active},i}$. The stage latency decomposes into computation and communication:

$$T^{(t)}_{\text{stage},i} = T^{(t)}_{\text{comp},i} + T^{(t)}_{\text{comm},i}, \tag{9}$$

where the computation latency is

$$T^{(t)}_{\text{comp},i} = \sum_{l \in \mathcal{L}^{(t)}_{\text{active},i}} (C_l \cdot r_i), \tag{10}$$

C_l is the fixed computational workload of layer l, and r_i is the per-layer processing time of device d_i obtained via offline profiling. $T^{(t)}_{\text{comm},i}$ is the measured latency for forwarding intermediate activations to the downstream device d_{i+1}. The pipeline throughput is bottlenecked by the slowest stage:

$$T^{(t)}_{\text{btk}} = \max_{i \in \{1,\ldots,N\}} \left\{ T^{(t)}_{\text{stage},i} \right\}. \tag{11}$$

We quantify pipeline load imbalance via the stage latency range:

$$\Delta T^{(t)} = \max_{i \in \{1,\ldots,N\}} \left\{ T^{(t)}_{\text{stage},i} \right\} - \min_{i \in \{1,\ldots,N\}} \left\{ T^{(t)}_{\text{stage},i} \right\}. \tag{12}$$

A larger $\Delta T^{(t)}$ indicates more severe pipeline bubbles and lower resource utilization. BAMLM aims to suppress $\Delta T^{(t)}$ below an acceptable threshold by dynamically migrating model layers across adjacent devices in response to real-time bandwidth fluctuations.

4.2 Sliding Window Monitoring

To filter instantaneous noise from bandwidth fluctuations, each edge device maintains a local window sequence H_i recording raw processing latency samples from consecutive historical batches, and adaptively adjusts its window length based on link jitter severity.

Upon obtaining the actual stage latency $T^{(t)}_{\text{stage},i}$ at batch t, the system first computes the historical mean $\mu^{(t-1)}_{H,i}$ over the previous window of length $W^{(t-1)}_i$ as a baseline:

$$\mu^{(t-1)}_{H,i} = \frac{1}{W^{(t-1)}_i} \sum_{j=1}^{W^{(t-1)}_i} T^{(t-j)}_{\text{stage},i}. \tag{13}$$

The latency fluctuation strength $V^{(t)}_i$ is then defined as the normalized deviation of the current sample from this baseline:

$$V^{(t)}_i = \frac{|T^{(t)}_{\text{stage},i} - \mu^{(t-1)}_{H,i}|}{\mu^{(t-1)}_{H,i}}. \tag{14}$$

A larger $V^{(t)}_i$ indicates more severe network jitter on the device's link. Based on this, the adaptive window length $W^{(t)}_i$ is updated via the following piecewise rule:

$$W^{(t)}_i = \begin{cases} W_{min}, & V^{(t)}_i \le \theta \\ \min\left(\lceil W_{min} + \lambda \cdot V^{(t)}_i \rceil,\ W_{max} \right), & V^{(t)}_i > \theta \end{cases}, \tag{15}$$

where W_{min} and W_{max} bound the adjustment range, θ is the jitter tolerance threshold, and λ is the window scaling factor. When fluctuation is within the normal range, a shorter window preserves sensitivity to recent bandwidth changes; when deviation exceeds the threshold, the window expands to incorporate more historical samples for statistical smoothing.

After determining $W^{(t)}_i$, the device appends $T^{(t)}_{\text{stage},i}$ to H_i and computes the smoothed stage latency:

$$\bar{T}^{(t)}_{\text{stage},i} = \frac{1}{W^{(t)}_i} \sum_{j=0}^{W^{(t)}_i - 1} T^{(t-j)}_{\text{stage},i}. \tag{16}$$

Each device periodically reports $\bar{T}^{(t)}_{\text{stage},i}$ to the coordinator node, which aggregates the global smoothed latencies to obtain a pipeline-wide performance view reflecting current load imbalance.

4.3 Model-Layer Migration Triggering

After collecting the smoothed stage latencies $\{\bar{T}^{(t)}_{\text{stage},1}, \ldots, \bar{T}^{(t)}_{\text{stage},N}\}$, the coordinator quantifies pipeline load imbalance via the global latency range:

$$\Delta\bar{T}^{(t)} = \max_{i\in\{1,\ldots,N\}} \{\bar{T}^{(t)}_{\text{stage},i}\} - \min_{i\in\{1,\ldots,N\}} \{\bar{T}^{(t)}_{\text{stage},i}\} \tag{17}$$

Since transient noise has been filtered by the sliding window, a large $\Delta\bar{T}^{(t)}$ reliably indicates persistent pipeline bubbles. However, as frequent physical weight transmission incurs communication overhead and disrupts inference context, the system tolerates minor imbalance by defining a dynamic triggering threshold proportional to the global average stage latency $\mu^{(t)}_{\text{sys}}$:

$$\mu^{(t)}_{\text{sys}} = \frac{1}{N}\sum_{i=1}^{N} \bar{T}^{(t)}_{\text{stage},i}, \qquad \Gamma^{(t)} = \gamma \cdot \mu^{(t)}_{\text{sys}} \tag{18}$$

where $\gamma \in (0, 1)$ is the preset steady-state tolerance ratio. Migration is triggered only when $\Delta\bar{T}^{(t)} > \Gamma^{(t)}$, indicating that load skew has severely restricted pipeline throughput.

To prevent system oscillation from repeated blind migrations—caused by stale historical data remaining in local sliding windows after a migration—the system introduces a cooldown mechanism. Let t_{last} be the batch index of the last migration and T_{cool} the cooldown period, which must exceed the maximum window length across all devices to ensure global observations are fully refreshed. The comprehensive migration trigger is:

$$\mathbb{I}^{(t)}_{\text{trigger}} = \begin{cases} 1, & \text{if } \Delta\bar{T}^{(t)} > \Gamma^{(t)} \text{ and } (t - t_{last}) > T_{cool} \\ 0, & \text{otherwise} \end{cases} \tag{19}$$

When $\mathbb{I}^{(t)}_{\text{trigger}} = 1$, the coordinator activates the migration decision module to recompute the optimal active layer set $\{\mathcal{L}^{(t+1)}_{\text{active},i}\}_{i=1}^{N}$ for each device.

4.3.1 Fine-Grained Dynamic Model-Layer Migration Mechanism. To address the challenge where the massive parameters of Vision-Language Models (VLMs) incur prohibitively high communication overheads for online weight migration under persistent system load imbalance—making edge deployment highly difficult—this section proposes a fine-grained dynamic model-layer migration mechanism. This mechanism decouples "boundary pre-partitioning" from "model-layer migration": in the offline phase, candidate boundaries are preset based on the available GPU memory of the devices; in the inference phase, the optimal set of layers to migrate is dynamically calculated based on real-time latency differences. This approach circumvents the prohibitive costs associated with traditional online parameter transmission, achieves fine-grained load balancing across the pipeline, and ultimately reduces inference latency.

4.3.2 Migration Boundary Pre-partitioning. This step follows the model partitioning algorithm and executes before weight distribution. To reserve elastic space for runtime migration, the system introduces a memory allocation ratio $\rho \in (0, 1)$, restricting the initial partitioning to at most $\rho \cdot M^i_{\text{total}}$ of device d_i's total GPU memory M^i_{total}. The truly available physical memory for hosting candidate layers is then:

$$M^{avail}_i = M^i_{\text{total}} - M^i_{\text{init}} - M_{\text{buffer}}, \tag{20}$$

where M^i_{init} covers static weights, initial KV cache, and intermediate activations of the initially assigned layer set $\mathcal{L}^{init}_i$, and M_{buffer} is a safety margin against GPU memory overflow.

At each boundary between adjacent devices d_i and d_{i+1}, the system delineates a shared candidate layer set $C_{i,i+1}$ that is preloaded onto both devices via redundant storage. Its maximum size is constrained by the minimum available memory of both parties:

$$k_{mem} = \min\left(\lfloor M^{avail}_i / m_{layer} \rfloor, \lfloor M^{avail}_{i+1} / m_{layer} \rfloor\right), \tag{21}$$

where m_{layer} is the average memory footprint of a single model layer. The system sets $|C_{i,i+1}| = k_{mem}$ and extracts the corresponding boundary layers from $\mathcal{L}^{init}_i$. The remaining layers form the base set $\mathcal{L}^{base}_i$—the core layers device d_i must execute independently under any network condition:

$$\mathcal{L}^{base}_i = \mathcal{L}^{init}_i \setminus (C_{i-1,i} \cup C_{i,i+1}). \tag{22}$$

The full weight set persistently stored by each device is thus:

$$\mathcal{W}_i = C_{i-1,i} \cup \mathcal{L}^{base}_i \cup C_{i,i+1}, \tag{23}$$

where the left candidate set of d_1 and the right candidate set of d_N are empty. During online inference, migration requires only ultra-low-latency control commands from the coordinator: by freezing candidate layers on the straggler device and activating the corresponding layers on the adjacent device, the computational boundary shifts without any physical weight transmission.

4.3.3 Model-Layer Migration. When $\mathbb{I}^{(t)}_{\text{trigger}} = 1$, the coordinator computes the globally optimal layer reorganization strategy. To prevent local offloading from inducing bottleneck shifts elsewhere, the system jointly adjusts candidate layer attribution at all $N - 1$ boundaries simultaneously.

For adjacent devices d_j and d_{j+1} ($j \in \{1, \ldots, N-1\}$), let $k_j \in \{0, 1, \ldots, |C_{j,j+1}|\}$ denote the number of layers in $C_{j,j+1}$ assigned to d_j, with the remaining $|C_{j,j+1}| - k_j$ layers activated by d_{j+1}. The global migration strategy is uniquely represented as $K = \{k_1, k_2, \ldots, k_{N-1}\}$. Given K, the reconstructed layer set $\mathcal{L}_i(K)$ of device d_i assembles from its base set and the allocated boundary subsets, yielding the expected computation latency:

$$\hat{T}_{comp,i}(K) = \sum_{l\in\mathcal{L}_i(K)} (C_l \cdot r_i). \tag{24}$$

For communication latency, since boundary shifts only alter the size of intermediate activations while steady-state network performance remains consistent over short periods, the expected communication latency is proportionally scaled from the smoothed estimate $\bar{T}^{(t)}_{\text{comm},i}$:

$$\hat{T}_{\text{comm},i}(K) = \bar{T}^{(t)}_{\text{comm},i} \cdot \frac{S_i(K)}{S^{(t)}_i}, \tag{25}$$

where $S^{(t)}_i$ and $S_i(K)$ are the current and post-migration activation sizes of device d_i, respectively. Let $\hat{T}_i(K) = \hat{T}_{\text{comp},i}(K) + \hat{T}_{\text{comm},i}(K)$. The system minimizes the global stage latency range via:

$$K^* = \arg\min_K \left(\max_{i\in\{1,\ldots,N\}} \hat{T}_i(K) - \min_{i\in\{1,\ldots,N\}} \hat{T}_i(K) \right). \tag{26}$$

The physical GPU memory constraints imposed during offline pre-partitioning significantly prune the solution space, enabling the coordinator to solve for K^* efficiently online.

5 Performance Evaluation

5.1 Experimental Setting

Experimental Environment. We construct a testbed cluster of five NVIDIA Jetson series edge devices exhibiting heterogeneity in both computational capability and memory capacity: one AGX Orin (64GB), two Orin NX (16GB), and two Orin Nano (8GB). All devices are interconnected via a network switch, with Linux TC (Traffic Control) employed to impose configurable bandwidth constraints on inter-node links, simulating weak-network conditions characteristic of real-world edge deployment. The inference system is built on NVIDIA JetPack 6.2 SDK and PyTorch, with all model weights loaded in FP16 precision to accommodate edge memory constraints.

The cluster is logically partitioned into a *coordinator node* and *compute nodes*. One Orin NX device serves as the coordinator, dedicated solely to scheduling, KV cache index maintenance, and control instruction dispatch, and is excluded from model inference computation. The remaining four devices—one AGX Orin, one Orin NX, and two Orin Nano—constitute the heterogeneous inference pipeline ($N = 4$), and are responsible for executing the VLM computational workload.

Hyperparameter Configuration. For the adaptive window adjustment mechanism, window size boundaries are set to $W_{\min} = 5$ and $W_{\max} = 20$, with fluctuation tolerance threshold $\theta = 0.1$ and scaling coefficient $\lambda = 2$. The load imbalance tolerance coefficient is $\gamma = 0.15$, and the post-migration cooldown period is $T_{cool} = 30$ batches. The initial memory allocation ratio is $\rho = 0.85$, reserving 15% of physical memory per device as an elastic buffer for candidate layer hosting, with a safety margin of $M_{buffer} = 512$ MB.

For the IHS module, the target iteration step count is $K_{step} = 8$, with per-iteration token bounds $B_{\max} = 2048$ and $B_{\min} = 32$. The KV cache safety waterline is set to $KV_{thresh} = 0.1$, permanently reserving 10% of available memory to prevent decoding interruptions from memory overflow.

Evaluation Metrics. The following five metrics are adopted to evaluate inference speed, accuracy, and resource overhead.

- **Average Inference Latency (AIL):** The average per-token generation time (ms/token) across all K test requests.
- **Inference Accuracy (ACC):** The proportion of correctly predicted samples over the test set.
- **Peak Memory Utilization:** The sum of peak GPU memory usage across all edge devices during inference, reflecting the cumulative footprint of model weights, KV cache, and intermediate activations at the cluster level.
- **Time to First Token (TTFT):** The elapsed time from request arrival to the generation of the first output token. TTFT mainly reflects queuing, prefill, image encoding, and initial pipeline execution latency.
- **Time Per Output Token (TPOT):** The average decoding time per output token after the first token is generated. For a request with L generated tokens, TPOT is computed as $(t_{\text{finish}} - t_{\text{first}})/(L-1)$.

Baseline Methods. Comparisons are conducted along two dimensions: *inference scheduling* (FasterTransformer, Orca, Sarathi-Serve) and *dynamic migration and distributed deployment* (LinguaLinked, DynaPipe).

- **FasterTransformer [19]:** A static batching baseline operating under first-come-first-served scheduling, which waits for all requests in a batch to complete before dispatching the next. It serves as the scheduling efficiency baseline.
- **Orca [29]:** An iteration-level scheduler that allows new requests to join and completed requests to leave at each iteration boundary, boosting throughput over static batching.
- **Sarathi-Serve [1]:** A hybrid batching method that decomposes long prefill tasks into sub-chunks co-executed with decoding, alleviating the blocking effect of prefill on decoding tasks.
- **LinguaLinked [33]:** A distributed inference system for heterogeneous edge devices that generates an overlap-based model allocation scheme offline and supports runtime task redistribution based on inter-node performance discrepancies.
- **DynaPipe [9]:** A dynamic pipeline method that adjusts layer assignments by predicting computation time deviations across stages, representing the traditional migration paradigm focused on computational overhead. Its performance under bandwidth-varying scenarios is examined to assess its handling of communication-sensitive imbalances.

Datasets. GYU-DET [12] contains 11,123 high-resolution industrial images across six surface defect categories, targeting feature extraction under complex visual conditions. We abbreviate GYU-DET as GYU in tables for compactness. VizWiz-VQA [6] contains 31,173 real-world image-question pairs, assessing model robustness across diverse open-domain scenes.

VLM Models. Qwen2.5-VL-32B [21] and Yi-VL-34B [28] are employed as backbone models. Each exceeds 30 billion parameters and requires over 60 GB of GPU memory under FP16 precision, enabling evaluation of load-aware partitioning and hybrid scheduling under heterogeneous edge constraints.

5.2 Analysis of Intra-Batch Scheduling (IHS) Performance

To validate the load adaptability of IHS under dynamic request workloads, we simulate Poisson-distributed request arrivals spanning sparse (1 req/s) to congested (5 req/s) scenarios. Using Qwen2.5-VL-32B under a 100 Mbps bandwidth constraint on both GYU-DET and VizWiz-VQA, we examine the impact of arrival rate λ on three metrics: Time to First Token (TTFT), Time Per Output Token (TPOT), and Average Inference Latency (AIL). Results are presented in Figure 4. IHS consistently outperforms the hybrid scheduling baseline Sarathi-Serve, with detailed analysis as follows.

(1) Time to First Token (TTFT). Under low-load conditions ($\lambda \leq 2$), IHS and Sarathi-Serve maintain comparably low TTFT values. As λ increases, Sarathi-Serve exhibits a pronounced upward trend, whereas IHS leverages its token flow control mechanism to dynamically regulate the prefill–decode ratio within each batch. At peak load ($\lambda = 5$), IHS stabilizes TTFT at 18.9 s, a notable reduction

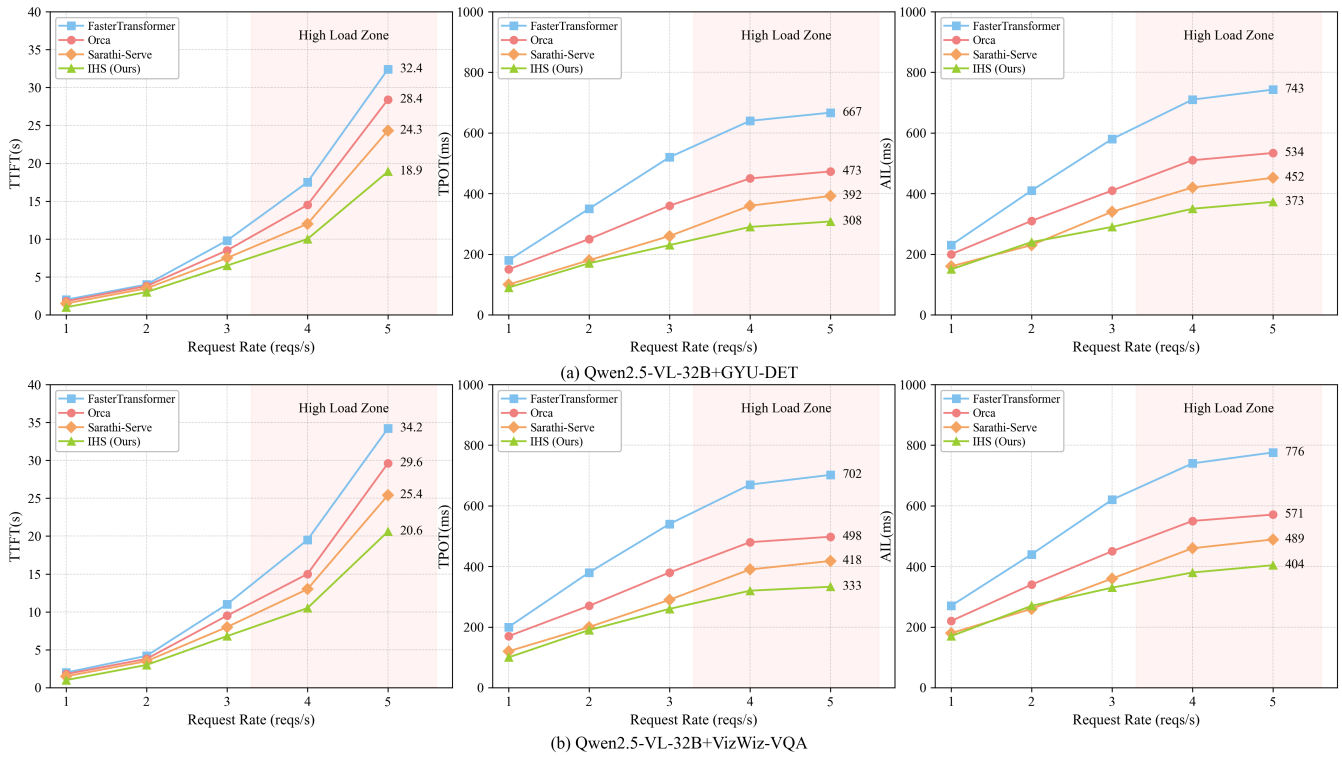


Figure 4: Performance Variation of Different Scheduling Methods Under Dynamic Load.

from Sarathi-Serve's 24.3 s, demonstrating effective suppression of first-token latency under heavy workloads.

(2) Time Per Output Token (TPOT). IHS reduces decoding latency by dynamically computing token quotas and evenly distributing prefill tokens across consecutive iteration steps. Under low load ($\lambda \leq 2$), both methods perform comparably. Under high-load congestion ($\lambda = 5$), IHS achieves approximately 20% lower TPOT than Sarathi-Serve, with more substantial gains over unscheduled baselines such as FasterTransformer.

(3) Average Inference Latency (AIL). IHS reduces end-to-end latency by jointly optimizing TTFT and TPOT. On GYU-DET under high load, IHS achieves an AIL of approximately 373 ms/token, outperforming Sarathi-Serve's 452 ms/token. Consistent gains are observed on VizWiz-VQA, which features longer output sequences, confirming that IHS reliably reduces average inference latency across diverse load-congestion scenarios.

5.3 Analysis of BAMLM Performance

5.3.1 Sensitivity Analysis of Steady-State Tolerance Threshold. Unless otherwise specified, the sensitivity analysis is conducted on GYU-DET. To identify the optimal steady-state tolerance threshold γ, we treat inter-node bandwidth as the environmental disturbance variable and sweep γ across $\{0.05, 0.10, 0.15, 0.20\}$, with results reported in Table 1. Average inference latency exhibits a consistent decrease-then-increase trend as γ grows, indicating a clear optimal range.

Table 1: Performance Comparison of Imbalance Tolerance Coefficients Under Different Models and Bandwidth Interference Levels.

Model	Bandwidth	Average Inference Latency (ms/token)			
		$\gamma = 0.05$	$\gamma = 0.10$	$\gamma = 0.15$	$\gamma = 0.20$
Qwen2.5-VL-32B	85 Mbps	328	**303**	306	314
	70 Mbps	366	338	**324**	386
	50 Mbps	387	354	**341**	404
Yi-VL-34B	85 Mbps	342	325	**318**	331
	70 Mbps	371	351	**340**	394
	50 Mbps	401	379	**367**	418

When $\gamma = 0.05$, the system becomes overly sensitive, frequently misclassifying minor bandwidth variations (e.g., 85 Mbps) as persistent imbalance and triggering unnecessary migrations, whose reconfiguration overhead significantly increases latency.

When $\gamma = 0.10$, the system reacts more aggressively to bandwidth variations. Although it achieves the best latency in one mild-interference case, its performance is less robust under stronger bandwidth degradation. In contrast, $\gamma = 0.15$ provides the best overall trade-off across most model-bandwidth combinations.

When $\gamma = 0.15$, the system achieves the optimal trade-off between sensitivity and rebalancing overhead, effectively filtering routine bandwidth noise while triggering migration only upon genuine steady-state violation. Under this configuration, Qwen2.5-VL-32B

Table 2: Inference Performance Comparison of Different Methods under Heterogeneous Bandwidth Scenarios.

Model	Method	Avg. Latency (ms/token)		Accuracy (%)		Memory (GB)	
		GYU	VizWiz	GYU	VizWiz	GYU	VizWiz
Qwen2.5-VL-32B	DynaPipe	376	392	74.8	75.4	**84.8**	**84.6**
	LinguaLinked	365	378	75.2	75.9	87.2	87.1
	BAMLM	**341**	**356**	**75.2**	**76.0**	89.4	89.2
Yi-VL-34B	DynaPipe	427	434	76.8	77.4	**87.1**	**87.2**
	LinguaLinked	396	401	**77.6**	**78.2**	88.5	88.6
	BAMLM	**367**	**374**	77.4	78.1	89.8	89.7

Note: Bold numbers indicate the best value in each column. For Avg. Latency, lower is better; for Accuracy, higher is better; for Memory, lower is better.

achieves an optimal average inference latency of 341 ms/token and Yi-VL-34B attains 367 ms/token under the 50 Mbps perturbation scenario, both outperforming all other γ settings.

When $\gamma = 0.20$, the system tolerates excessive imbalance: under severe degradation (50 Mbps or below), the trigger threshold is never met, causing pipeline bubbles to accumulate and throughput to degrade substantially.

Accordingly, $\gamma = 0.15$ is adopted as the baseline threshold across all subsequent BAMLM experiments.

5.3.2 Inference Performance Analysis Under Heterogeneous Bandwidth. To validate BAMLM under dynamic bandwidth evolution, we evaluate Qwen2.5-VL-32B and Yi-VL-34B on both GYU-DET and VizWiz-VQA. The experiment initializes all pipeline links at 100 Mbps; at the 50th inference iteration, bandwidth between intermediate nodes is abruptly reduced to approximately 50 Mbps via Linux TC, after which the system operates continuously under this heterogeneous condition. Comparative results across average inference latency, accuracy, and peak GPU memory are reported in Table 2.

(1) Average Inference Latency. BAMLM achieves superior environmental robustness over all baselines. DynaPipe's computation-focused scheduling logic overlooks the asymmetric communication delays induced by Linux TC, resulting in delayed decisions and limited effectiveness. LinguaLinked's coarse-grained device-level partitioning tends to overload downstream nodes during migration, triggering localized congestion. In contrast, BAMLM precisely captures link-level latency variations via SWBS and leverages FGMLM to smoothly migrate layers from congested to bandwidth-sufficient adjacent stages, achieving an optimal average inference latency of 341 ms/token on Qwen2.5-VL-32B and 367 ms/token on Yi-VL-34B under the 50 Mbps heterogeneous bandwidth scenario.

(2) Inference Accuracy. Inference accuracy remains comparable across all methods and datasets, with only marginal variation. Yi-VL-34B achieves a slightly higher accuracy range (76.8%–78.2%) than Qwen2.5-VL-32B. Since no method alters model weights or architecture, the results confirm that BAMLM substantially reduces inference latency while preserving output correctness.

(3) Peak GPU Memory Consumption. BAMLM incurs a modestly higher peak GPU memory footprint than other methods, stemming from its redundant residency mechanism that preloads candidate layer parameters from adjacent stages onto each node. Although this incurs a moderate memory increment, it enables migration via logical operator routing alone—without real-time cross-node weight transmission—eliminating the communication overhead that migration would otherwise impose. Experimental results confirm that memory consumption remains within hardware tolerance, validating the trade-off of reserving memory to eliminate weight transmission latency.

Table 3: Overall Inference Performance Comparison of IHS-LM and Baselines.

Model	Method	Avg. Latency (ms/token)		Accuracy (%)		Memory (GB)	
		GYU	VizWiz	GYU	VizWiz	GYU	VizWiz
Qwen2.5-VL-32B	Baseline	376	392	74.8	75.4	**84.8**	**84.6**
	w/o BAMLM	358	371	75.1	75.8	89.3	89.1
	w/o IHS	341	356	75.2	76.0	89.4	89.2
	IHS-LM	**328**	**342**	**75.2**	**76.1**	89.5	89.3
Yi-VL-34B	Baseline	427	434	76.8	77.4	**87.1**	**87.2**
	w/o BAMLM	389	398	77.3	77.9	89.7	89.6
	w/o IHS	367	374	77.4	78.1	89.8	89.7
	IHS-LM	**352**	**361**	**77.5**	**78.3**	89.9	89.8

Note: Baseline refers to the combination of Sarathi-Serve and DynaPipe. The "w/o IHS" variant replaces IHS with Sarathi-Serve while retaining BAMLM, whereas the "w/o BAMLM" variant replaces BAMLM with DynaPipe while retaining IHS. Bold numbers indicate the best value in each column. For Avg. Latency and Memory, lower is better; for Accuracy, higher is better.

5.4 Overall Performance Evaluation of IHS-LM

To evaluate the combined effect of intra-batch hybrid scheduling (IHS) and bandwidth-aware model layer migration (BAMLM), we conduct experiments on the full IHS-LM system, which integrates both modules. The system is tested under dynamic bandwidth conditions to simulate real-world edge environments. We compare IHS-LM against a hybrid baseline (Sarathi-Serve + DynaPipe), and two ablation configurations: one without IHS (w/o IHS) and one without BAMLM (w/o BAMLM). Experiments are conducted on two datasets, GYU and VizWiz, under high-load and heterogeneous bandwidth scenarios, and the average inference latency (AIL) is measured in ms per token. This setup allows us to quantify both the individual contributions of each module and the overall benefit of combining IHS and BAMLM in heterogeneous edge environments.

As shown in Table 3, IHS-LM achieves the lowest inference latency on both GYU and VizWiz datasets, outperforming all baseline and partial configurations in inference latency. Specifically, IHS-LM reduces average inference latency by up to 17.5% compared with the Baseline configuration, while preserving model accuracy. These results demonstrate the effectiveness of jointly optimizing intra-batch scheduling and stage-level layer migration for pipeline-parallel inference on heterogeneous edge devices.

The experimental results in Table 3 show that the superior performance of IHS-LM comes from the complementary effects of its two core modules.

The intra-batch hybrid scheduling (IHS) module mitigates intra-batch workload imbalance by dynamically adjusting the prefill–decode

token ratio based on token distribution and available KV cache capacity. This ensures that all micro-batches within a batch complete in a similar time, improving overall pipeline efficiency.

The bandwidth-aware model layer migration (BAMLM) module removes inter-stage imbalance by dynamically migrating model layers across devices according to real-time bandwidth variations. This ensures that all pipeline stages finish their workloads in a balanced manner.

6 Related Work

Generative Model Inference Scheduling. LLM inference involves a compute-intensive Prefill phase and a memory-intensive Decode phase [23, 31], whose divergent resource demands have motivated efficient scheduling strategies. FasterTransformer [19] uses request-level static batching, causing underutilization when output lengths vary. Orca [29] introduces iteration-level dynamic batching, improving memory use and concurrency. Sarathi-Serve [1] mitigates prefill–decode interference via chunked prefilling and stall-free scheduling. TD-Pipe [30] decouples Prefill and Decode through phase separation and micro-batch task stealing, reducing scheduling overhead.

However, these approaches cannot adapt the prefill–decode token ratio based on KV cache occupancy and real-time token distribution, leaving intra-batch imbalance in distributed edge inference. IHS addresses this via fine-grained token flow control that monitors pending tokens and KV cache vacancy [11], maintaining a balanced prefill–decode ratio and ensuring consistent computational load across all pipeline stages.

Dynamic Load Balancing for Pipeline Inference. Static partitioning [8, 17] fails under runtime bandwidth fluctuations, especially in heterogeneous edge environments [26, 27, 32]. Existing solutions fall into two groups.

Communication-oriented methods, such as COACH [5] and Helix [16], optimize data flow within fixed partitions, but cannot fully resolve stage imbalance under severe bandwidth degradation. Layer migration methods, e.g., LinguaLinked [33] and DynaPipe [9], adjust layer-to-device assignments at runtime, yet often ignore link bandwidth variation or the prohibitive cost of large-scale weight retransmission in multi-hop edge networks. BAMLM addresses both limitations by combining adaptive bandwidth sensing with offline candidate pre-partitioning, enabling precise migration without excessive weight retransmission overhead.

7 Conclusion

In this paper, we presented IHS-LM, a low-latency pipeline-parallel inference framework for VLM serving on heterogeneous edge devices. To address intra-batch workload imbalance, we introduce the Intra-batch Hybrid Scheduling (IHS) module, which dynamically regulates the prefill–decode token ratio via fine-grained token flow control. To tackle pipeline stage imbalance under bandwidth fluctuations, we proposed the Bandwidth-Aware Model Layer Migration (BAMLM) module, combining reliable imbalance detection with low-overhead layer migration. Experiments on a real five-device NVIDIA Jetson cluster demonstrate that IHS-LM reduces average inference latency by up to 17.5% over state-of-the-art baselines under heterogeneous bandwidth conditions, while preserving accuracy and respecting memory constraints. In future work, we plan to explore hybrid parallelism strategies and adaptive partitioning schemes under highly dynamic edge network conditions.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China under Grant 62572171; in part by the 16th Batch of Science and Technology Development Programs (Innovation Consortium Projects) of Suzhou under Grant LHT202404; in part by the Technology Talent and Platform Program of Yunnan Province under Grant 202405AK340002; in part by the Technology Project of Huaneng Group under Grant HNKJ24-H167; and in part by the High Performance Computing Platform, Hohai University.

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 2024*. USENIX Association, 117–134.

[2] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, and Yuxiong He. 2022. DeepSpeed-Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 46:1–46:15. doi:10.1109/SC41404.2022.00051

[3] Fenglong Cai, Dong Yuan, Zhe Yang, and Lizhen Cui. 2024. Edge-LLM: A Collaborative Framework for Large Language Model Serving in Edge Computing. In *IEEE International Conference on Web Services, ICWS 2024, Shenzhen, China, July 7-13, 2024*. IEEE, 799–809. doi:10.1109/ICWS62655.2024.00099

[4] Esha Choukse, Pratyush Patel, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, Rodrigo Fonseca, and Ricardo Bianchini. 2025. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. *IEEE Micro* 45, 4 (2025), 54–59. doi:10.1109/MM.2025.3575361

[5] Luyao Gao, Jianchun Liu, Hongli Xu, Sun Xu, Qianpiao Ma, and Liusheng Huang. 2025. Accelerating End-Cloud Collaborative Inference via Near Bubble-free Pipeline Optimization. In *IEEE INFOCOM 2025 - IEEE Conference on Computer Communications*. IEEE, 1–10. doi:10.1109/INFOCOM55648.2025.11044632

[6] Danna Gurari, Qing Li, Abigale J. Stangl, Anhong Guo, Chi Lin, Kristen Grauman, Jiebo Luo, and Jeffrey P. Bigham. 2018. VizWiz Grand Challenge: Answering Visual Questions From Blind People. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018*. 3608–3617. doi:10.1109/CVPR.2018.00380

[7] Chaoyang He, Shen Li, Mahdi Soltanolkotabi, and Salman Avestimehr. 2021. PipeTransformer: Automated Elastic Pipelining for Distributed Training of Large-scale Models. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event (Proceedings of Machine Learning Research)*, Marina Meila and Tong Zhang (Eds.). PMLR, 4150–4159. http://proceedings.mlr.press/v139/he21a.html

[8] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Advances in Neural Information Processing Systems 32, NeurIPS 2019*. 103–112.

[9] Chenyu Jiang, Zhen Jia, Shuai Zheng, Yida Wang, and Chuan Wu. 2024. DynaPipe: Optimizing Multi-task Training through Dynamic Pipelines. In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22-25, 2024*. ACM, 542–559. doi:10.1145/3627703.3629585

[10] Yizhang Jin, Jian Li, Tianjun Gu, Yexin Liu, Bo Zhao, Jinxiang Lai, Zhenye Gan, Yabiao Wang, Chengjie Wang, Xin Tan, and Lizhuang Ma. 2025. Efficient multimodal large language models: a survey. *Vis. Intell.* 3, 1 (2025). doi:10.1007/S44267-025-00099-6

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

[12] Ruiping Li, Linchang Zhao, Hao Wei, Guoqing Hu, Yongchi Xu, Bocheng Ouyang, and Jin Tan. 2025. Multi-defect type beam bridge dataset: GYU-DET. *Scientific Data* 12, 1 (2025), 1101. doi:10.1038/s41597-025-05395-w

[13] Han Liang, Jiahui Zhou, Zicheng Zhou, Xiaoxi Zhang, and Xu Chen. 2025. STADI: Fine-Grained Step-Patch Diffusion Parallelism for Heterogeneous GPUs. *CoRR* abs/2509.04719 (2025). arXiv:2509.04719 doi:10.48550/ARXIV.2509.04719

[14] Xing Liu, Lizhuo Luo, Ming Tang, and Chao Huang. 2025. FlowSpec: Continuous Pipelined Speculative Decoding for Efficient Distributed LLM Inference. *CoRR* abs/2507.02620 (2025). arXiv:2507.02620 doi:10.48550/ARXIV.2507.02620

[15] Davide Macario, Hulya Seferoglu, and Erdem Koyuncu. 2025. Model-Distributed Inference for Large Language Models at the Edge. In *31st IEEE International Symposium on Local and Metropolitan Area Networks, LANMAN 2025, Lille, France, July 7-8, 2025*. IEEE, 1–6. doi:10.1109/LANMAN66415.2025.11154542

[16] Yixuan Mei, Yonghao Zhuang, Xupeng Miao, Juncheng Yang, Zhihao Jia, and Rashmi Vinayak. 2025. Helix: Serving Large Language Models over Heterogeneous GPUs and Network via Max-Flow. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2025*. ACM, 586–602. doi:10.1145/3669940.3707215

[17] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019*. ACM, 1–15. doi:10.1145/3341301.3359646

[18] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient large-scale language model training on GPU clusters using Megatron-LM. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021*. ACM, 58. doi:10.1145/3458817.3476209

[19] NVIDIA Corporation. 2019. FasterTransformer. https://github.com/NVIDIA/FasterTransformer.

[20] 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. *CoRR* abs/2407.00079 (2024). arXiv:2407.00079 doi:10.48550/ARXIV.2407.00079

[21] Qwen Team, Alibaba Group. 2025. Qwen2.5-VL: Technical Report. *CoRR* abs/2502.13923 (2025). arXiv:2502.13923 doi:10.48550/arXiv.2502.13923

[22] Xiaofei Wang, Yiwen Han, Victor C. M. Leung, Dusit Niyato, Xueqiang Yan, and Xu Chen. 2020. Convergence of Edge Computing and Deep Learning: A Comprehensive Survey. *IEEE Commun. Surv. Tutorials* 22, 2 (2020), 869–904. doi:10.1109/COMST.2020.2970550

[23] Jiayang Wu, Wensheng Gan, Zefeng Chen, Shicheng Wan, and Philip S. Yu. 2023. Multimodal Large Language Models: A Survey. In *IEEE International Conference on Big Data, BigData 2023, Sorrento, Italy, December 15-18, 2023*. IEEE, 2247–2256. doi:10.1109/BigData59044.2023.10386743

[24] Jiacheng Yang, Jun Wu, Yaoyao Ding, Zhiying Xu, Yida Wang, and Gennady Pekhimenko. 2026. SwiftFusion: Scalable Sequence Parallelism for Distributed Inference of Diffusion Transformers on GPUs. In *Proceedings of the ACM Conference on AI and Agentic Systems, CAIS 2026, San Jose, CA, USA, May 26-29, 2026*. ACM, 1037–1050. doi:10.1145/3786335.3813174

[25] Shengyuan Ye, Jiangsu Du, Liekang Zeng, Wenzhong Ou, Xiaowen Chu, Yutong Lu, and Xu Chen. 2024. Galaxy: A Resource-Efficient Collaborative Edge AI System for In-situ Transformer Inference. In *IEEE INFOCOM 2024 - IEEE Conference on Computer Communications, Vancouver, BC, Canada, May 20-23, 2024*. IEEE, 1001–1010. doi:10.1109/INFOCOM52122.2024.10621342

[26] Shengyuan Ye, Bei Ouyang, Liekang Zeng, Tianyi Qian, Xiaowen Chu, Jian Tang, and Xu Chen. 2025. Jupiter: Fast and Resource-Efficient Collaborative Inference of Generative LLMs on Edge Devices. In *IEEE INFOCOM 2025 - IEEE Conference on Computer Communications*. IEEE, 1–10. doi:10.1109/INFOCOM55648.2025.11044734

[27] JinYi Yoon, Yeongsin Byeon, Jeewoon Kim, and HyungJune Lee. 2022. EdgePipe: Tailoring Pipeline Parallelism With Deep Neural Networks for Volatile Wireless Edge Devices. *IEEE Internet Things J.* 9, 14 (2022), 11633–11647. doi:10.1109/JIOT.2021.3131407

[28] Alex Young, Bei Chen, Chao Li, et al. 2024. Yi: Open Foundation Models by 01.AI. *CoRR* abs/2403.04652 (2024). arXiv:2403.04652 doi:10.48550/arXiv.2403.04652

[29] 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 2022*. USENIX Association, 521–538.

[30] Hongbin Zhang, Taosheng Wei, Zhenyi Zheng, Jiangsu Du, Zhiguang Chen, and Yutong Lu. 2025. TD-Pipe: Temporally-Disaggregated Pipeline Parallelism Architecture for High-Throughput LLM Inference. In *Proceedings of the 54th International Conference on Parallel Processing, ICPP 2025*. ACM, 689–698. doi:10.1145/3754598.3754621

[31] Jingyi Zhang, Jiaxing Huang, Sheng Jin, and Shijian Lu. 2024. Vision-Language Models for Vision Tasks: A Survey. *IEEE Trans. Pattern Anal. Mach. Intell.* 46, 8 (2024), 5625–5644. doi:10.1109/TPAMI.2024.3369699

[32] Mingjin Zhang, Xiaoming Shen, Jiannong Cao, Zeyang Cui, and Shan Jiang. 2025. EdgeShard: Efficient LLM Inference via Collaborative Edge Computing. *IEEE Internet Things J.* 12, 10 (2025), 13119–13131. doi:10.1109/JIOT.2024.3524255

[33] Junchen Zhao, Yurun Song, Simeng Liu, Ian G. Harris, and Sangeetha Abdu Jyothi. 2024. LinguaLinked: Distributed Large Language Model Inference on Mobile Devices. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics, ACL 2024*. Association for Computational Linguistics, 160–171. doi:10.18653/V1/2024.ACL-DEMOS.16

[34] Zhi Zhou, Xu Chen, En Li, Liekang Zeng, Ke Luo, and Junshan Zhang. 2019. Edge Intelligence: Paving the Last Mile of Artificial Intelligence With Edge Computing. *Proc. IEEE* 107, 8 (2019), 1738–1762. doi:10.1109/JPROC.2019.2918951

[35] Bingjie Zhu, Zhixiong Chen, Liqiang Zhao, Hyundong Shin, and Arumugam Nallanathan. 2025. Efficient LLM Inference over Heterogeneous Edge Networks with Speculative Decoding. *CoRR* abs/2510.11331 (2025). arXiv:2510.11331 doi:10.48550/ARXIV.2510.11331

[36] Kan Zhu, Yufei Gao, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, Ziren Wang, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. 2025. NanoFlow: Towards Optimal Large Language Model Serving Throughput. In *19th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2025, Boston, MA, USA, July 7-9, 2025*, Lidong Zhou and Yuanyuan Zhou (Eds.). USENIX Association, 749–765. https://www.usenix.org/conference/osdi25/presentation/zhu-kan