Maestro: Workload-Aware Cross-Cluster Scheduling for LLM-Based Multi-Agent Systems

Jinghao Wang[1], Xiao Zhou[1], Xiaoyang Sun[2†], Yihui Zhang[1], Yilong Li[1], Tianyu Wo[1], Xu Wang[1], Chunming Hu[1], Renyu Yang[1]
[1]Beihang University [2]University of Leeds
{wang_jinghao, zhouxiao2021, zhangyihui, liyiloong, woty, xuwang, hucm, renyuyang}@buaa.edu.cn; x.sun4@leeds.ac.uk

***Abstract*—Large Language Model based Multi-Agent Systems (LLM-MAS) have emerged as a powerful paradigm for tackling complex tasks by breaking them into collaborative workflows of specialized LLM-powered agents. However, deploying such multi-agent workloads at scale poses significant system challenges. Each user query spawns an iterative pipeline of LLM calls, greatly amplifying resource consumption compared to single-turn queries. In resource-constrained cloud settings, these workflows face non-deterministic and input-dependent costs at decode stage, heavy-tailed multi-model requirements with memory fragmentation and over-provisioning, and cross-cluster scheduling trade-offs. We present Maestro, a workload-aware scheduling system designed for LLM-MAS serving under strict GPU budgets. Maestro explicitly leverages agent semantics and roles: it predicts the output length and memory usage of each stage and uses this prediction to drive a hierarchical scheduler. At the node level, Maestro enables dynamic multi-model co-location via hierarchical weight caching and elastic memory provisioning. At the cluster level, it performs latency-aware routing to avoid cold-start delays and memory overloads. At the global level, it enforces workflow-aware prioritization to minimize head-of-line blocking for interactive tasks. Across prototype experiments and trace-driven simulations, Maestro reduces KV-reservation HBM by 67.2% and improves high-contention SLO attainment over EDF by 23.6 percentage points.**

***Index Terms*—Large Language Model, Multi-Agent System, Resource Scheduling, Cloud Computing, Workload Prediction.**

I. Introduction

Large Language Models (LLMs) are reshaping modern AI services [5, 11]. With recent open-source models (e.g., LLaMA [32], DeepSeek [11], Qwen [39]) demonstrating exceptional general-purpose reasoning and generation capabilities, *LLM-based Multi-Agent Systems* (LLM-MAS) have emerged as a promising paradigm for tackling complex tasks [14, 38]. In this architecture, a user request is decomposed into a structured workflow where specialized agents (e.g., planner, solver, critic) collaborate via natural language messages. This collaborative approach integrates memory, planning, and communication modules, enabling enhanced autonomy and adaptability that often outperform monolithic models in both robustness and quality [8, 12, 34].

Running LLM-MAS at scale introduces fundamental challenges for inference serving infrastructure, particularly in private or on-premises deployments that lack the near-infinite elasticity of public clouds. In such resource-constrained settings [29, 31], GPU capacity is typically constrained, fragmented, and distributed. These constraints impose a critical trade-off between Quality-of-Service (QoS) and resource efficiency–a conflict significantly exacerbated by the *dependency-coupled* nature of multi-agent workflows. Consider a sequential workflow (e.g., Planner → Coder → Reviewer): keeping paused agents' context resident in GPU memory ensures low latency but blocks other requests, while reclaiming it to free capacity incurs reloading overheads that violate SLOs. Consequently, naive scheduling [37] or static provisioning–effective for independent queries–fails to resolve this magnified contention in LLM-MAS.

Challenge 1: Non-deterministic and input-dependent execution cost. LLM inference is autoregressive and produces outputs of variable and hard-predictable length. In a multi-agent workflow, the latency and memory footprint of each stage depend on its output, which is unknown in advance. Without accurate cost estimates, the scheduler cannot properly prioritize requests or allocate memory, leading to head-of-line (HoL) blocking where long reasoning steps delay short ones. This phenomenon is further exacerbated in LLM-MAS, where decoding costs vary with factors such as agent's role, type of service/tool, and reasoning strategy (e.g., Chain-of-Thought).

Challenge 2: Long-tail usage patterns and memory contention. In multi-agent pipelines, model invocation typically follows a heavy-tailed pattern: only a few models are invoked frequently, whereas most are used rarely. Assigning a dedicated GPU to each model results in resource underutilization, while loading models on demand incurs cold-start costs (e.g., weight loading) that breach interactive latency SLOs. Co-locating multiple models on a single GPU mitigates underutilization but intensifies memory contention. Beyond static model parameters, the KV cache size grows with output length, creating dynamic pressure on limited GPU memory.

Challenge 3: Cross-cluster routing trade-offs (latency vs. resource readiness). In distributed settings, GPU clusters differ in both network delay and resource availability. Routing to the topologically nearest cluster minimizes network latency, but that cluster may lack the cached model weights or sufficient GPU memory to serve the request immediately. A more distant cluster may demonstrate high *resource readiness* (i.e., the model is loaded and memory is available), but the higher round-trip time increases end-to-end latency, especially time-

†Dr. Xiaoyang Sun is the corresponding author

to-first-token. Thus, inefficient routing can propagate delays throughout the pipeline, slowing all subsequent stages.

Existing solutions only partially address these challenges [9, 25, 37]: prediction-aware schedulers (FASTSERVE [37]) and multi-model systems (MuxServe [9], QLM [25]) target single-turn queries with fixed partitioning; cross-cluster frameworks (SkyServe [22], AIBrix [31]) follow static routing oblivious to model readiness or KV affinity. No existing system couples low-overhead stage-level cost estimation with cross-cluster allocation for agentic workflows.

To bridge this gap, we propose Maestro, a workload-aware cross-cluster scheduling system for LLM-MAS. Unlike existing serving stacks that see only opaque LLM requests, Maestro characterizes the workload along five agent-level dimensions that directly shape resource behavior: the agent's *role and workflow position*, its *tool-invocation intent* for the current step, the *predicted per-stage output length and KV footprint*, the job's *remaining workflow time*, and the *model readiness* of each candidate cluster. Driven by this agent-level view of the workload, Maestro coordinates decisions across three tiers: at the *node level*, memory-safe multi-model colocation via hierarchical weight residency (GPU-CPU-disk) and prediction-guided elastic KV allocation; at the *cluster level*, fitness-based dispatch that balances network latency, model readiness, and KV feasibility; and at the *global level*, workflow-aware preemptive Shortest-Remaining-Time-First queueing that protects latency-critical stages without starving long analytic jobs.

This paper makes the following contributions.

i) Agent-aware cost prediction. Maestro presents a semantic- and structure-aware stage cost predictor that distinguishes concise tool-invocation outputs from extended Chain-of-Thought generations, lowering token-length prediction error by about 19% and yielding accurate per-stage runtime and tool-usage estimates for scheduling.

ii) Multi-agent spatio-temporal multiplexing. Maestro implements a node-level runtime that co-optimizes hierarchical model placement across GPU/CPU/disk and prediction-driven elastic KV cache allocation, allowing memory-efficient colocation of many specialized models under strict GPU memory.

iii) Feasibility-aware cross-cluster scheduling. Maestro unifies network latency, per-cluster model readiness, and predicted KV footprint into a single fitness score that drives cross-cluster dispatch, coupled with boundary-preemptive Shortest-Remaining-Time-First to curb dependency-induced blocking under mixed SLOs.

iv) Implementation and evaluation. Through a complete system prototype on a physical GPU cluster and trace-driven simulations, Maestro reduces prediction MAE by 19.2%, reduces KV-reservation HBM by 67.2%, and improves high-contention SLO attainment by 23.6 percentage points.

II. BACKGROUND & MOTIVATION

Multi-agent serving workflows under mixed SLOs. LLM-MAS processes each user request as a structured workflow of dependent stages $\{T_1, \ldots, T_n\}$. Each stage T represents a logical step executed by a specialized agent using a specific model

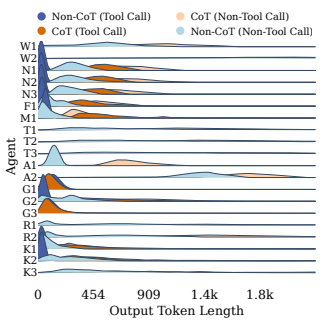

Fig. 1. Output-token length distributions under non-CoT and CoT settings, tool-call and non-tool-call.

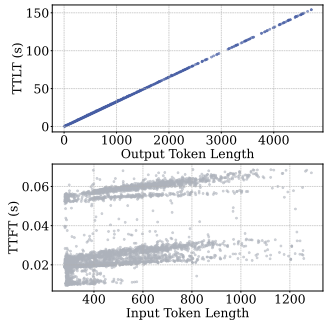

Fig. 2. Relationships between output token length and TTLT (latency), and input token length and TTFT.

$M(T)$, which may involve natural language generation or external tool invocation [5, 14, 38, 34]. Dependency coupling means upstream delays propagate to block subsequent steps, while resource costs vary systematically by agent role–e.g., concise tool calls versus extended Chain-of-Thought reasoning [41].

In LLM-MAS with SLOs, *user-facing stages* prioritize Time-to-First-Token (TTFT) for interactivity, whereas *internal reasoning steps* demand low Time-to-Last-Token (TTLT) to rapidly unblock downstream dependencies. Scheduling is distinctively challenging: dependency-induced head-of-line blocking [25] means slow internal stages delay the final response, and cross-cluster routing must balance network latency against resource readiness [22, 31]. Unlike stateless microservices [10], LLM-MAS requires jointly managing these heterogeneous urgencies and heavy, stateful memory footprints (KV cache) under strict constraints.

LLM inference cost model. An LLM (especially a decoder-only model) processes each request in two phases: a *prefill* (or prompt processing) phase and an autoregressive *decode* phase. Let P be the prompt length (i.e., number of input tokens) and L be the output length (number of generated tokens). The time for the prefill $T_{\text{prefill}}(P)$ increases roughly with P, and the time for decoding rises with L. For large L, the decode time mainly dominates the total latency. The TTLT can be described as $T_{\text{TTLT}} = T_{\text{prefill}}(P) + L \cdot t_{\text{decode}}$, where t_{decode} is the average per-token generation time for the model on given hardware. This means that if we underestimate how long L will be, we risk underestimating the runtime of the job.

KV cache memory pressure. At inference, memory is consumed not only by model weights but also by the KV cache, which stores hidden states for all processed tokens. KV cache usage scales linearly with the total sequence length $(P+L)$, amplified by model architectural factors (e.g., layers, heads) and precision. This couples runtime and memory costs: generating a long output occupies GPU compute longer while consuming proportionally more memory, and under high concurrency the aggregate KV footprint can quickly exceed physical GPU limits.

State-of-the-art serving engines [16, 42, 50] employ paging techniques to optimize memory layout. While PagedAtten-

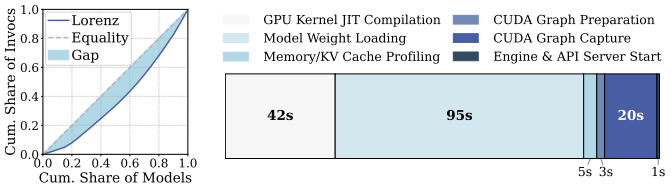

Fig. 3. Model-invocation long tail (left) from Chatbot Arena traces [49]. Cold-start breakdown (right) for an 8B model under standard serving.

tion [16] effectively eliminates internal fragmentation via non-contiguous allocation, it does not resolve the fundamental *capacity constraint*. Under heavy load with long-context agents, the aggregate KV footprint can still exceed physical GPU limits. Consequently, the system is forced to perform strict admission control, blocking new requests or preempting active ones, rather than relying on low-level memory compaction.

Multi-agent deployments exacerbate memory challenges. LLM-MAS typically utilizes specialized model variants for different agent roles. To avoid cold-start latency, serving systems often maintain multiple models in a *resident state* (i.e., weights loaded in GPU or host memory). While this ensures model readiness, it incurs strict resource costs: each resident model occupies memory not only for its parameters but also for associated runtime overheads (e.g., CUDA graphs, context buffers). This static footprint competes directly with the dynamic capacity required for the KV cache.

Motivation. To concretize these challenges, we performed detailed measurements on representative multi-agent workloads.

Observation-1: *Agent roles, tool invocation, and reasoning mode strongly determine output length and latency.* Output token distributions vary widely across workflow stages, shown in Figure 1: tool-oriented agents typically emit short, structured outputs, while user-facing or reasoning-heavy agents generate long free-form text. Enabling Chain-of-Thought reasoning further shifts output toward heavy-tailed distributions. We observe that output length strongly correlates with TTLT (inference latency), while prompt length mainly affects TTFT, shown in Figure 2. Consequently, accurate per-stage output-length prediction is essential to anticipate long-running stages, mitigate head-of-line blocking, and perform memory-safe admission, since KV cache usage scales with sequence length.

Observation-2: *Multi-model demand is highly skewed, and cold starts fundamentally conflict with latency SLOs.* Workload traces exhibit a pronounced long tail, shown in Figure 3. Many specialized models are invoked infrequently but remain necessary. Static per-model GPU allocation wastes capacity, while on-demand loading incurs cold-start delays of tens of seconds, far exceeding interactive SLOs [49]. Naive multi-model colocation further exacerbates GPU memory contention and fragmentation, motivating predictive, hierarchical model residency and elastic memory management.

Observation-3: *Cross-cluster routing must balance network latency with resource readiness.* Although routing to nearby clusters minimizes network round-trip time (RTT), Figure 4 implies that local clusters may lack the required model weights

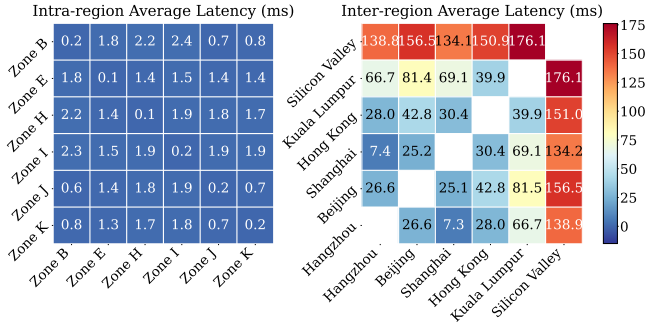

Fig. 4. Intra- and inter-region network latency reported by [2].

or sufficient KV capacity. In such scenarios, dispatching execution to a remote cluster with a *ready* model yields lower end-to-end latency, provided that the inter-region RTT (typically tens to hundreds of milliseconds) is orders of magnitude smaller than the local cold-start overhead (tens of seconds). Consequently, effective scheduling must jointly optimize for network proximity and *resource readiness*–specifically, both model residency and KV availability.

In summary, these observations indicate that high-performance LLM-MAS serving hinges on (i) precise workload forecasting at each stage, (ii) coordinated multi-model GPU memory orchestration, and (iii) intelligent, feasibility-aware cluster-wide scheduling. Maestro meets these needs through a unified and prediction-guided scheduling pipeline.

III. System Design

Maestro enables LLM-MAS execution in multi-cluster environments by jointly deciding *when* and *where* to place each workflow stage T, selecting cluster and GPU while managing model residency and GPU memory.

A. Overview

Maestro executes LLM-MAS through a stage-driven closed-loop control pipeline, as illustrated in Figure 5. Each workflow stage T is an independent scheduling unit that passes through five phases:

Agent-context observation. When a stage is created, Maestro intercepts the invocation and extracts a compact descriptor capturing the agent role, workflow position, tool availability, and a *semantic embedding* of the input context. In multi-cluster deployments, it also records the source cluster and any policy constraints that restrict placement.

Cost prediction. A lightweight predictor at the *dispatch gateway* estimates the expected output length $\hat{L}(T)$, the KV cache requirement $\hat{R}_{\mathrm{kv}}(T)$, and the tool-use probability $\hat{p}_{\mathrm{tool}}(T)$, incurring negligible latency relative to network RTT.

Scheduling decision. The global scheduler filters infeasible nodes by policy and memory constraints, then dispatches the stage to the node that maximizes a fitness score balancing network latency, time-to-start, and KV cache feasibility. Stages are ordered by workflow-aware Shortest-Remaining-Time-First with preemption for latency-critical work.

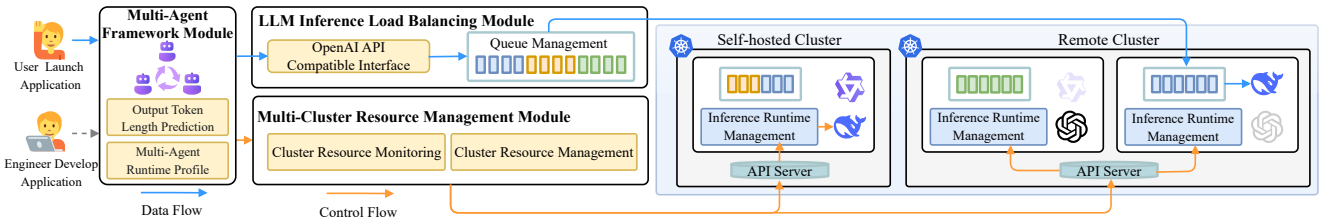

Fig. 5. The system architecture and workflow of Maestro.

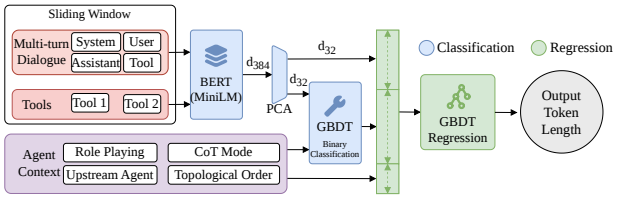

Fig. 6. Agent-aware output-length prediction architecture in Maestro.

Node-level execution and memory management. The runtime coordinates model loading, KV cache allocation with explicit memory accounting, and minimum-disruption reclamation under memory pressure, reporting disruption costs back to the scheduler.

Post-execution profiling. Actual output length, latency, and memory usage are recorded to incrementally calibrate the predictor and re-prioritize remaining stages.

B. Agent-Aware Cost Prediction

Feature representation. Input features $\boldsymbol{x}(T)$ concatenate *structured features* (including agent role, workflow position, invocation index, and tool availability) and *semantic features* derived from the input text. To handle long prompts efficiently without the overhead of full-scale attention, Maestro applies a sliding-window encoding using a lightweight BERT model (MiniLM) and aggregates embeddings via mean pooling. This yields a compact semantic representation $\boldsymbol{x}_{\text{sem}}(T)$ that captures context complexity.

Two-phase prediction pipeline. We observe that tool-invoking phases typically produce short structured outputs, while user-facing stages generate longer free-form text, resulting in a bimodal output-length distribution. This motivates a two-stage predictor (Figure 6) that explicitly models tool-invoking intent.

Maestro first employs a lightweight classifier to estimate the probability that stage T triggers a tool call:

$$\hat{p}_{\text{tool}}(T) = \Pr[\text{tool}(T) = 1 \mid \boldsymbol{x}(T)] \tag{1}$$

To ensure the predicted confidence aligns with empirical frequencies, we apply isotonic regression to calibrate the classifier outputs, providing a reliable continuous signal to guide downstream estimation. If no tools are available, we set $\hat{p}_{\text{tool}}(T) = 0$.

Maestro then predicts the output length $\hat{L}(T)$ using structured features $\boldsymbol{x}_{\text{str}}(T)$, semantic features $\boldsymbol{x}_{\text{sem}}(T)$, and the predicted tool-invoking probability $\hat{p}_{\text{tool}}(T)$. To capture role-specific generation patterns, we train per-role regressors when training data are available; otherwise, we fall back to a shared global model. To mitigate the impact of heavy-tailed output distributions, we perform a regression on $\log(1 + L(T))$ (where $L(T)$ is the length of the ground-truth output) and apply the inverse transform at inference time.

Translation to system metrics. Maestro translates the predicted length into execution-time and memory estimates using calibrated per-model microbenchmarks. Let $t_{\text{pre}}(P, M)$ be the prefill latency for prompt length P and $t_{\text{dec}}(M)$ be the average per-token decode latency. The estimated execution time is:

$$\hat{T}_{\text{exec}}(T) = t_{\text{pre}}(P(T), M(T)) + t_{\text{dec}}(M(T)) \cdot \hat{L}(T) \tag{2}$$

which captures the inference cost. Tool execution latency is profiled as an external workflow stage and added to the remaining-time profile when traces expose it; the current runtime, however, does not reserve CPU or network resources for external tools. The KV cache requirement is estimated as:

$$\hat{R}_{\text{kv}}(T) = \alpha(M(T)) \cdot \big(P(T) + \hat{L}(T)\big) \tag{3}$$

where $\alpha(M)$ denotes the model-specific memory footprint per token. These estimates provide the scheduler with explicit signals for execution-time ordering and memory-feasible placement.

C. Node-Level Execution and Memory Management

Maestro employs a node-level runtime manager to enable *memory-feasible multi-model colocation* on a single GPU. The design combines *temporal multiplexing*, which amortizes model activation costs, with *spatial multiplexing*, which increases KV cache concurrency under a shared GPU memory budget. Crucially, the runtime exports readiness and KV-feasibility signals that allow the cross-cluster scheduler to avoid routing latency-sensitive stages to nodes that would incur cold starts or disruptive memory admissions.

Model readiness abstraction and memory accounting. Maestro employs a *model readiness abstraction* to capture the activation cost of a model on a node. A model may be in one of five states: *Running* (weights and execution context resident on GPU), *Sleeping* (weights offloaded to host memory, but lightweight GPU runtime contexts–e.g., CUDA graphs and JIT kernels–are retained to accelerate reloading), *CPU-resident* (weights cached in host memory without preserved GPU context), *Disk-resident*, or *Remote*.

Algorithm 1: Hierarchical Weight Residency and Eviction

Input: Model m, Capacities $C_{\text{gpu}}, C_{\text{cpu}}, C_{\text{disk}}$
Output: Model m becomes GPU-ready

$S_m \leftarrow \text{GetModelSize}(m)$;
$Loc \leftarrow \text{LocateModel}(m)$;
if $Loc = \text{GPU}$ **then**
  $\text{LRU.Update}(\text{GPU}, m)$;
  return Success;
while $\text{UsedGPU}() + S_m > C_{\text{gpu}}$ **do**
  $v \leftarrow \text{LRU.PeekLast}(\text{GPU})$;
  $\text{OffloadToHost}(v)$;
  $\text{LRU.Remove}(\text{GPU}, v)$;
  $\text{LRU.Update}(\text{CPU}, v)$;
if $Loc \in \{\text{Disk}, \text{Remote}\}$ **then**
  while $\text{UsedCPU}() + S_m > C_{\text{cpu}}$ **do**
    $v \leftarrow \text{LRU.PeekLast}(\text{CPU})$;
    $S_v \leftarrow \text{GetModelSize}(v)$;
    if $\text{UsedDisk}() + S_v \leq C_{\text{disk}}$ **then**
      $\text{MoveToDisk}(v)$;
    else
      $\text{DeleteFromDisk}(v)$;
    $\text{LRU.Remove}(\text{CPU}, v)$;
  $\text{LoadToCPU}(m)$;
$\text{LoadToGPU}(m)$;
$\text{LRU.Update}(\text{GPU}, m)$;
return Success;

For each model, the runtime reports both its readiness state and an activation latency estimated via a *profiled bandwidth model* (i.e., $T_{\text{act}} \approx \text{Size}/\text{BW}_{\text{tier}}$), allowing switching costs to be compared uniformly across clusters.

Multi-model serving is constrained by a shared GPU memory budget in which persistent warm contexts directly compete with KV cache capacity. Rather than reserving a fixed fraction of memory for KV cache, Maestro enforces *explicit memory accounting* with admission-time feasibility checks.

Let M_{total} denote the total GPU memory available to the runtime, M_{kv} the current KV cache usage, $\mathcal{S}$ the set of warm models (Running or Sleeping), M^k_{ctx} the persistent context footprint of model k, and M_{other} non-model overheads. The reserved non-KV footprint is $M_{\text{res}} = \sum_{k \in \mathcal{S}} M^k_{\text{ctx}} + M_{\text{other}}$ and memory safety is enforced by $M_{\text{kv}} + M_{\text{res}} \leq M_{\text{total}}$.

At admission time, the runtime checks whether the incoming stage's additional KV demand can be safely allocated. If feasible, the stage is admitted and the KV admission headroom $R^{\text{head}}_{\text{kv}}(N) = M_{\text{total}} - M_{\text{res}} - M_{\text{kv}}$ is reported as a scheduling signal; otherwise, the runtime rejects the stage or triggers minimum-impact memory coordination, reporting infeasibility and the associated disruption cost.

Temporal multiplexing via hierarchical weight residency. To amortize cold-start overheads, the runtime maintains a hierarchical weight residency path (GPU $\rightarrow$ Host RAM $\rightarrow$ Local Disk $\rightarrow$ Remote Storage), with LRU eviction policies applied at each tier. When a stage requests model m, the runtime activates it from the fastest available tier. If the GPU memory is full, the least recently used model is offloaded to host memory (transitioning to *Sleeping* or *CPU-resident* state). If host memory is subsequently insufficient, cold weights are evicted to local disk (or discarded if disk capacity is exceeded) to make room. Algorithm 1 summarizes this cascading load-and-evict process.

Algorithm 2: Minimum-Impact Memory Coordination

Input: Required memory R, Resident engines $\mathcal{E}$
Output: Degradation plan P, Interrupt flag I_{active}

$M_{\text{freed}} \leftarrow 0; P \leftarrow [\,]; I_{\text{active}} \leftarrow \text{False}$;
$\mathcal{E}_{sorted} \leftarrow \text{SortByPriority}(\mathcal{E})$;
for $e \in \mathcal{E}_{sorted}$ **do**
  if $M_{\text{freed}} \geq R$ **then**
    break;
  $Action \leftarrow \text{DetermineBestAction}(e, P)$;
  if $Action \in \{\text{Preempt}, \text{Abort}\}$ **then**
    $I_{\text{active}} \leftarrow \text{True}$;
  $M_{\text{freed}} \leftarrow M_{\text{freed}} + \text{EstimateFreedMemory}(e, Action)$;
  $P.\text{append}((e, Action))$;
if $M_{\text{freed}} < R$ **then**
  return Failure;
return P, I_{active};

The runtime periodically reports the current warm-set composition and model activation-cost estimates. The global scheduler exploits these signals during cross-cluster routing, preferentially dispatching latency-sensitive stages to clusters where the required model is already warm.

Spatial multiplexing with virtual-memory KV cache. Temporal multiplexing alone necessitates conservative KV admission, which can underutilize GPU memory at low load. In nodes supporting CUDA virtual memory management (VMM), Maestro enables *spatial multiplexing* by constructing a shared virtual KV cache pool and mapping KV pages on demand. We integrate *kvcached* [44] to allocate and reclaim KV pages via CUDA VMM, reducing fragmentation and enabling elastic KV provisioning. To maintain reclaimability under multi-model colocation, long-lived prefix caching is disabled in this mode unless explicitly required.

Although the virtual KV address space may exceed physical memory, Maestro preserves safety through admission control and on-demand mapping–if physical allocation fails, the runtime rejects the stage or triggers controlled degradation and reports infeasibility to the scheduler, preventing out-of-memory failures. The runtime advertises VMM support as a capability-conditioned signal, which the scheduler treats as a hard feasibility constraint for high-concurrency stages and a soft preference for latency-sensitive interactive stages.

Minimum-impact memory coordination as a routing signal. When KV admission fails, the runtime executes Algorithm 2 to derive a *minimum-impact degradation plan*. We define five degradation levels with ascending disruption costs: (1) transitioning *Idle-Running* models to *Sleeping*; (2) evicting *Sleeping* models; (3) stopping pending sleep transitions; (4) swapping out KV for *Active* models; and (5) aborting *Active* executions.

The algorithm adopts a greedy strategy, iterating through resident engines $\mathcal{E}$ sorted by priority (Idle $\rightarrow$ Sleeping $\rightarrow$ Active) and accumulating freed memory via state-dependent actions until the requirement R is met.

The runtime calculates total disruption penalty $C_{\text{deg}}(N, T)$ as the *aggregate restoration latency* required to execute a plan:

$$C_{\text{deg}}(N, T) = \sum_{(e,a) \in P} c(e, a) + \mathbf{1}[I_{\text{active}}] \cdot c_{\text{int}} \tag{4}$$

Here, $c(e, a)$ is derived from profiled storage bandwidth (for model reloading) or compute throughput (for KV regeneration), while c_{int} corresponds to the SLO violation threshold.

D. Workload-Aware Cross-Cluster Scheduling

Maestro employs a cross-cluster scheduling control loop that jointly performs routing, admission, and queueing for multi-agent workflow stages. Each placement must satisfy policy constraints, KV cache feasibility, and latency objectives. The scheduler uses (i) a feasibility-aware, fitness-based routing policy to select *where* a stage runs, and (ii) a profile-driven remaining-time queueing policy to decide *when* stages are dispatched under workflow dependencies.

Cross-cluster fitness. For an arriving stage T, Maestro predicts the KV demand $\hat{R}_{\text{kv}}(T)$ and applies a safety margin $R_{\text{need}}(T) = (1+\rho) \cdot \hat{R}_{\text{kv}}(T)$. Rather than a fixed heuristic, ρ is set from recent prediction errors: we maintain an EWMA of the relative underestimation $e = \max(0, R_{\text{kv}}/\hat{R}_{\text{kv}} - 1)$ and choose ρ as a high quantile (e.g., 90–95%) of e within a sliding window; in practice it typically falls in $[0.1, 0.3]$. The scheduler first filters candidate nodes using routing policies and feasibility signals reported by the runtime manager; only nodes that can safely admit $R_{\text{need}}(T)$ under memory constraints are considered.

Among feasible candidates, Maestro selects the node that minimizes the expected stage completion latency $\widehat{T}_{\text{e2e}}(N,T) \approx \text{RTT}(\text{src}(T), N) + T_{\text{ready}}(N,T) + \widehat{T}_{\text{exec}}(T) + \eta\, C_{\text{deg}}(N,T)$. Since $\widehat{T}_{\text{exec}}(T)$ is node-invariant for a fixed model-engine pair, it does not affect the routing argmin and is used in the queueing policy below. We therefore rank nodes by the equivalent compact score:

$$S(N,T) = A(N,T) - \lambda\, T_{\text{ready}}(N,T) - \mu\, C_{\text{deg}}(N,T) \quad (5)$$

where $T_{\text{ready}}(N,T)$ is the expected time-to-start on node N and $C_{\text{deg}}(N,T)$ is the disruption penalty reported by the runtime (Eq. 4). The affinity term $A(N,T)$ combines (i) network proximity using a decreasing transform of the measured RTT and (ii) KV cache fit via best-fit packing based on the runtime-reported headroom $R^{\text{head}}_{\text{kv}}(N)$ (larger stages prefer nodes whose headroom is close to $R_{\text{need}}(T)$ among feasible candidates); for latency-sensitive interactive stages, we increase the weight on the network component. Throughout, we apply robust min–max normalization with per-metric 5/95-percentile bounds over a recent window (clipped outside) to prevent outliers from dominating the score, and we report T_{ready} and C_{deg} in milliseconds (default λ=μ=1).

The ready time term decomposes into queueing and activation latency:

$$T_{\text{ready}}(N,T) = T_{\text{q}}(N,T) + T_{\text{act}}(N, M(T)) \quad (6)$$

where $T_{\text{q}}(N,T)$ is the estimated node-local queueing delay (smoothed with EWMA), and $T_{\text{act}}(N, M(T))$ is the model activation latency determined by the model readiness state on node N.

Profile-driven remaining-time queueing. Maestro employs a two-level queueing architecture: a global queue that determines the dispatch order between jobs and node-local queues that schedule execution after placement. To mitigate dependency-induced head-of-line blocking, the global queue orders jobs by an estimate of their remaining workflow execution time. For a job J currently executing stage T_k, the remaining time is estimated as

$$\widehat{T}_{\text{rem}}(J,k) = \widehat{T}_{\text{exec}}(T_k) + \widehat{T}_{\text{future}}(J,k) \quad (7)$$

where $\widehat{T}_{\text{exec}}(T_k)$ is derived from the predicted output length $\hat{L}(T_k)$ by Eq. 2. To account for branching and iterative behavior, Maestro maintains a rolling execution profile for each workflow template and estimates the future term using a conditional median over recent executions:

$$\widehat{T}_{\text{future}}(J,k) \approx T^{(0.50)}_{\text{next}}(\text{state}(J,k)) \quad (8)$$

where $\text{state}(J,k)$ summarizes the workflow position using the agent role, stage template, invocation index, and a discretized tool-invoking intent score.

Algorithm 3: Cross-Cluster Scheduling

Input: Stage T, candidate nodes $\mathcal{N}$ (across clusters)
Output: Selected node N^*

$(\hat{L}, \hat{R}_{\text{kv}}) \leftarrow \text{Predict}(T)$;
$R_{\text{need}} \leftarrow (1+\rho) \cdot \hat{R}_{\text{kv}}$;
$\mathcal{N}' \leftarrow \text{FilterByPolicyAndFeasibility}(\mathcal{N}, T, R_{\text{need}})$;
$(\lambda, \mu) \leftarrow \text{GetWeights}(T.\text{class})$;
$N^* \leftarrow \varnothing$; $S^* \leftarrow -\infty$;
for $N \in \mathcal{N}'$ **do**
  $A \leftarrow \text{Affinity}(\text{RTT}(\text{src}(T), N), R^{\text{head}}_{\text{kv}}(N), R_{\text{need}}, T.\text{class})$;
  $(T_{\text{ready}}, C_{\text{deg}}) \leftarrow \text{RuntimeEstimate}(N, T)$;
  $S \leftarrow A - \lambda \cdot T_{\text{ready}} - \mu \cdot C_{\text{deg}}$;
  if $S > S^*$ **then**
    $S^* \leftarrow S$; $N^* \leftarrow N$
return N^*;

Preemption and mixed SLOs. Under contention, Maestro preempts background stages only at stage boundaries (between LLM invocations) and re-queues them, avoiding disruption of in-flight decoding. This choice is intentionally conservative: token- or iteration-level preemption can further reduce blocking for exceptionally long decoding phases, but requires tighter integration with the decoding engine and KV migration path. In our design, the common preemption path only updates scheduler metadata and preserves KV allocations; when memory pressure prevents preservation, the recovery latency is charged through the degradation term C_{deg} in Eq. 4. To prevent oscillation due to estimation noise (e.g., in T_{ready}), we apply hysteresis: preemption is triggered only when the predicted latency gain exceeds a threshold and a per-job cooldown expires; queueing-delay estimates are smoothed with EWMA. KV cache allocations are preserved when feasible and otherwise reclaimed by the runtime using minimum-impact coordination, while aging gradually increases the effective priority of long-waiting background jobs to prevent starvation.

IV. Evaluation

We evaluate Maestro using both a physical GPU cluster and trace-driven simulations. We organize the results into three layers: (i) *overall performance*, assessing end-to-end

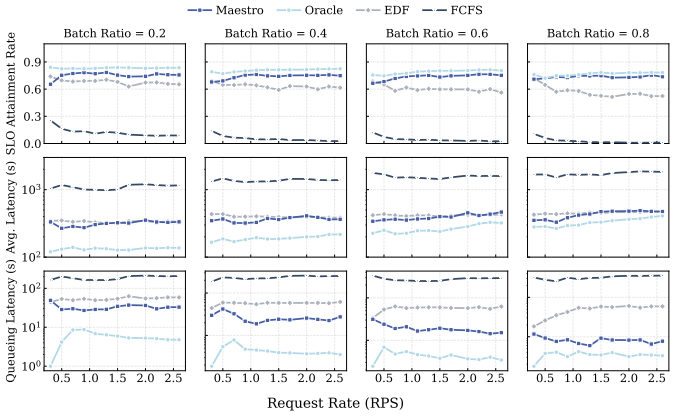

Fig. 7. Overall scheduling results across arrival rates and batch ratios: SLO attainment, mean latency, and interactive queueing delay.

TABLE I
WORKLOAD TRACE SUMMARY (I INDICATES INTERACTIVE, B INDICATES BATCH).

App.	Type	#Jobs	Input Source
Meeting Booking	I	8626	Synthetic
Document Writing	B	8319	Synthetic
News Collection	B	6616	Synthetic
Performance	B	6548	IBM HR Analytics [30]
QA Assistant	I	5849	MATH [13]/PM [3]/GPQA [28]/SQuAD [27]
Text Translation	B	5124	OpenCSG Chinese Corpus [45]
Food Assistant	I	3334	Synthetic
Travel Assistant	I	1543	Synthetic
Code Refactoring	B	810	LongBench v2 [4]
Total	–	**46,769**	**144,524 stages**

latency and SLO attainment under mixed workloads; (ii) *resource efficiency*, quantifying GPU cost savings and memory overcommitment enabled by multi-model colocation; and (iii) *component analysis*, evaluating the accuracy and runtime overhead of the agent-aware prediction module.

A. Experimental Setup

Testbed. We deploy Maestro in a multi-cluster environment. Our physical testbed comprises 32 GPU servers (each with 2 NVIDIA A100 GPUs, 128 CPU cores, 1TB RAM, and 2TB NVMe SSD local storage). The software stack is Ubuntu 22.04.5 with Kubernetes (v1.31.2), NVIDIA Driver 580.105.08, and CUDA 13.0. We use vLLM (v0.11.0) as the inference backend and extend its PagedAttention KV management with our elastic paging mechanism.

Workloads. We replay a multi-agent workflow trace collected from 9 representative LLM-MAS applications (4 interactive and 5 batch-style), covering serial, parallel, loop, and supervisor-worker patterns. The trace contains 46,769 jobs and 144,524 LLM-invocation stages (Table I). Both synthetic and public-dataset workloads are generated from fixed application templates encoding common agent topologies (e.g., serial tool-use loops, supervisor-worker fan-out/fan-in, multi-step reasoning with refinement); they differ only in input sources, ensuring that train and test jobs share application logic but use distinct prompts.

Baselines. We compare Maestro against the following baselines.

i) *Prediction*. We compare our two-stage *Maestro-Pred* (tool-intent + length regression) with: a) *Linear* (prompt-length-only regression); b) *BERT-MLP* [26] (semantic embedding + MLP, single-stage); and c) *Magnus* [6] (semantic embedding + regression). We train the predictor on recorded stage-level traces and evaluate it using a stratified temporal holdout: within each agent, tool-use, and thinking-mode group, the latest 20% of records are used as the test set and earlier records are used for training. For LightGBM regressors, the training records are further split by the same strata, using the latest 15% as a validation set for early stopping; all reported prediction numbers are measured on the held-out test records.

ii) *Scheduling*. We implement: a) *FCFS* (global FIFO); b) *EDF* (deadline-first for batch jobs, class-priority for interactive stages); and c) *Oracle-SRTF* (shortest true remaining time with perfect knowledge), as an upper bound. All scheduling baselines share the same vLLM backend, measured model profiles, dynamic batching support, and workload arrivals; they differ only in admission, routing, and queue ordering.

iii) *Multi-model serving*. We compare against: (a) *No-Colocation* (one model per GPU); and (b) *QLM-style Switching* [25] (multi-model with process-level restart/reload). We use QLM-style switching because it is the closest public baseline for multi-model SLO-oriented serving; No-Colocation represents the fully warm upper-cost configuration, while Maestro evaluates the benefit of retaining warm contexts without dedicating one GPU per model.

SLOs. We evaluate mixed SLOs consistent with agentic applications. For each workflow template, we first profile isolated executions and use the observed completion-time distribution to set workload-specific deadlines; batch-style jobs use end-to-end completion deadlines, while interactive jobs emphasize responsiveness. Since scheduling primarily affects time-to-start (queueing + activation) rather than per-token decode speed, we approximate user-perceived delay by the sum of per-stage waiting times, and report both interactive queueing delay and SLO attainment accordingly. The same SLO targets are used for all baselines.

B. Overall Performance

We first evaluate end-to-end performance under mixed-SLO workloads using trace-driven simulation. The request arrival rate is varied from low load ($\lambda = 0.4$ req/s) to high load ($\lambda = 2.0$ req/s), and the batch ratio from 0.20 to 0.80. Figure 7 reports SLO attainment, mean latency, and interactive queueing delay.

Under high load and large batch ratios, *FCFS* collapses due to severe head-of-line blocking, reducing SLO attainment to below 10%. *EDF* improves performance by prioritizing imminent deadlines but degrades when long jobs approach deadlines, causing short interactive stages to be delayed. In a representative high-stress configuration ($\lambda = 2.0$, batch ratio=0.8), EDF achieves 50.0% SLO attainment, whereas Maestro achieves 73.6%, corresponding to a 23.6 percentage points improvement. Maestro further reduces interactive

TABLE II
ABLATION STUDY OF THE PREEMPTION IN MAESTRO

#Nodes	Maestro		Maestro w/o Preempt	
	SLO ↑	Delay (ms) ↓	SLO ↑	Delay (ms) ↓
1	29%	538k	29%	1500k
2	**60%**	**389**	27%	235k
3	**38%**	**22**	26%	82k
4	**75%**	**9**	59%	19k
5	**85%**	**2**	84%	11k

TABLE III
TOOL-INTENT CLASSIFICATION COMPARISON (↑ HIGHER IS BETTER; ↓ LOWER IS BETTER).

Model	AUC ↑ MSE ↓	F1 Macro ↑ LogLoss ↓	Acc. ↑ False Recall ↑
Maestro-Pred	**0.9625** **0.0728**	0.8999 **0.2437**	0.8999 0.8966
MLP_64_32	0.9535 0.0827	0.8838 0.2837	0.8840 **0.9115**
MLP_128_64	0.9520 0.0834	0.8802 0.2948	0.8802 0.8906
MLP_BayesOpt_3	0.9609 0.0736	**0.9022** 0.3750	**0.9022** 0.8981
ResNet	0.9484 0.0909	0.8838 0.3327	0.8840 **0.9115**
TabTransformer	0.9502 0.0868	0.8848 0.2882	0.8848 0.8801

queueing delay by 84.8% relative to EDF, highlighting the effectiveness of remaining-time-aware prioritization.

We further evaluate the impact of preemption under extreme load ($\lambda = 5.0$, batch ratio=0.6) by comparing Maestro with and without preemption. Table II reports SLO attainment and interactive queueing delay as the number of nodes increases. Preemption mitigates head-of-line blocking from long-running batch stages: it raises SLO attainment on two and four nodes (60% vs. 27%; 75% vs. 59%), and still cuts interactive queueing delay by orders of magnitude even when SLOs align (2 ms vs. 11 s on five nodes). These results use boundary preemption (between LLM invocations); the common overhead is limited to priority checking and re-queueing, while resume costs from reclaimed state are reflected in the scheduler's degradation-cost term (Eq. 4).

C. Resource Efficiency

We evaluate the impact of multi-model colocation on GPU efficiency and responsiveness using (i) replay of real multi-model workflows and (ii) node-level utilization and memory-accounting analysis.

1) Cost-latency trade-off under tight GPU budgets: We evaluate application-level performance using a real multi-model *Travel Assistant* workflow comprising six LLM invocations across three models. Table IV reports end-to-end completion time under varying GPU budgets. With three GPUs, both approaches match the performance of exclusive

TABLE IV
TRAVEL ASSISTANT COMPLETION TIME VS. GPU BUDGET.

Method	1 GPU (s) ↓	2 GPUs (s) ↓	3 GPUs (s) ↓
Maestro	**92.4**	**84.9**	82.4
QLM	309.1	139.1	82.4

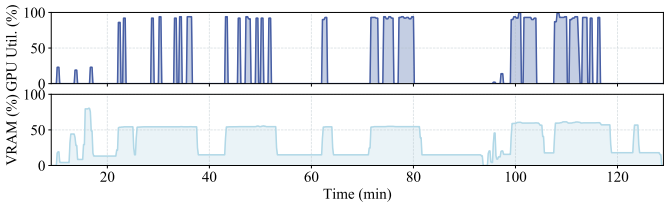

Fig. 8. GPU utilization and memory usage under multi-model colocation.

deployment. As the GPU budget decreases, QLM incurs substantial cold-start overheads, whereas Maestro maintains responsiveness, reducing completion time by 38.9% with two GPUs and 70.0% with one GPU. Relative to exclusive deployment, Maestro incurs modest completion-time increases (3.0% with two GPUs and 12.1% with one GPU) while achieving significant GPU cost reductions (33.3% and 66.7%, respectively), demonstrating a favorable cost-latency trade-off.

2) Utilization and memory overcommitment under colocation: Figure 8 illustrates GPU utilization and memory usage when multiple Qwen3 models are colocated on a single A100 GPU under low load. GPU utilization remains highly intermittent, indicating that exclusive deployment would waste substantial capacity. In contrast, Maestro maintains models in warm states and safely overcommits memory, enabling rapid response without dedicating separate GPUs to each model.

Table V reports runtime memory accounting for colocating five models. The total virtual KV-cache budget reaches approximately 122GB on a 40GB GPU, corresponding to a 3.05× effective memory footprint, i.e., 205% overcommitment or 67.2% less HBM than non-overcommitted reservation. The preserved CUDA-graph/runtime contexts consume 194–286MB per model (about 1.15GB in total for this five-model configuration), so the sleeping mechanism is not free; this footprint is explicitly counted in M_{res} and traded against activation latency. This level of overcommitment is feasible because actual KV usage is elastic and admission is guided by predicted demand, enabling substantial cost savings in low-load, long-tail multi-model scenarios.

D. Component Analysis

We conclude by analyzing the key system components that contribute to the end-to-end performance.

1) Tool-calling intent classification: Tool-invoking stages form a distinct execution mode with short, structured outputs in agent workflows, motivating explicit intent prediction. We compare the first-stage classifier of Maestro-Pred (LightGBM with combined semantic and structured features) against neural baselines, including MLP variants and dual-tower/Transformer-style fusion models. Table III reports eval-

TABLE V
MEMORY ACCOUNTING UNDER FIVE-MODEL COLOCATION ON A SINGLE A100 40GB GPU.

Model	CUDA Graph	Weight Size	Virtual KV Cache
Qwen3-0.6B	194 MB	1.12 GB	34.24 GB
Qwen3-1.7B	194 MB	3.21 GB	32.15 GB
Qwen3-4B	256 MB	7.55 GB	27.81 GB
Qwen3-8B	245 MB	15.27 GB	20.07 GB
Qwen3-14B	286 MB	27.52 GB	7.74 GB

TABLE VI
OUTPUT TOKEN LENGTH PREDICTION ACCURACY.

Metric	**Maestro-Pred**	Magnus	BERT-MLP	Linear
MAE ↓	**165.43**	204.74	239.33	496.86
R^2 ↑	**0.7774**	0.6721	0.5620	-0.3177

uation metrics including AUC, macro-F1, accuracy, MSE, log loss, and negative-class recall.

Maestro-Pred achieves the highest AUC (0.9625) and the lowest MSE (0.0728) and log loss (0.2437); the three-layer MLP gains marginally on F1/accuracy but at much higher log loss, i.e., worse calibration—a decisive property because the classifier output feeds the second-stage regressor as a continuous feature.

2) Output-length regression and ablation: Table VI reports output-length prediction accuracy. *Maestro-Pred* achieves an MAE of 165.43 tokens with an R^2 of 0.7774, reducing MAE by 19.2% relative to Magnus and by 30.9% relative to BERT-MLP. In contrast, the linear baseline yields a negative R^2, indicating that output length in multi-agent workflows is highly non-linear and cannot be captured by simple correlations with input length alone.

We evaluate two ablations: (i) Maestro-Pred w/o C removes the tool-calling intent classifier and directly regresses output length; (ii) Maestro-Pred w/o BERT further removes semantic embeddings, relying solely on structured agent features. As shown in Table VII, semantic embeddings contribute substantially, with overall R^2 decreasing from 0.7774 to 0.7129 without BERT. Tool-calling intent classification is most beneficial in non-CoT settings, where tool-invoking and non-tool-invoking stages exhibit sharper length divergence; under CoT, this benefit diminishes as both modes include explicit reasoning segments.

3) Online overhead: Prediction latency directly affects time-to-first-token. We measure end-to-end inference latency of the prediction module, including BERT encoding. Figure 9 reports P50/P95 latency for BERT-based methods. *Maestro-Pred* achieves a median latency of 11.24ms, comparable to BERT-MLP and significantly lower than Magnus (15.76ms). Adding the first-stage intent classifier incurs negligible overhead, as both stages share feature extraction and preprocessing.

4) Runtime Support for Colocation: Model Activation Latency: Figure 10 compares model startup and activation latency for models ranging from 0.6B to 14B parameters on a single A100 40GB GPU. Unlike QLM-style approaches that require process-level restarts for model switching [25],

TABLE VII
ABLATION STUDY OF THE PREDICTION MODULE IN MAESTRO.

Version	MAE ↓	R^2 ↑	MAE (CoT) ↓	MAE (non-CoT) ↓
Full	**165.43**	**0.7774**	265.97	**134.17**
w/o C	170.78	0.7722	**264.25**	141.72
w/o BERT	175.46	0.7129	288.18	140.42

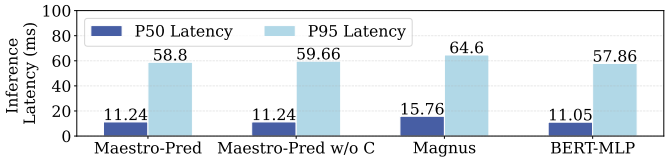

Fig. 9. Prediction overhead (P50/P95) for methods with BERT encoders.

Maestro reduces activation latency by reusing runtime contexts and avoiding process restarts, enabling practical multi-model serving under long-tail demand.

5) Cross-Cluster Dispatch and Node Fitness Scoring: To evaluate cross-cluster scheduling, we configure the simulator with a hybrid topology comprising three local and two remote nodes. The physical cluster validates node-level execution and colocation, while this simulator isolates cross-cluster routing effects using the measured model profiles and RTT matrix from Figure 4. We compare three policies: (i) Baseline, which performs simple load balancing without prediction-guided bin packing; (ii) BinPack Only, which enables KV-demand-aware bin packing but ignores network latency ($\gamma = 0$); and (iii) Maestro-Aff, which applies the full fitness score with a latency weight ($\gamma = 0.25$) to penalize remote placements. Table VIII reports the average interactive queueing delay under varying arrival rates λ. Fitness weights are selected on the validation split and kept fixed across test workloads; robust normalization in Eq. 5 prevents any single metric from dominating under outlier RTT or activation estimates.

KV-demand-aware bin packing substantially reduces queueing delay by mitigating fragmentation and consolidating compatible requests, achieving a 58.3% reduction relative to Baseline under low load ($\lambda = 0.5$) and remaining effective under high load ($\lambda = 2.0$). Incorporating network latency in Maestro-Aff further improves performance by favoring local execution for latency-sensitive stages, demonstrating the effectiveness of prediction-guided cross-cluster scheduling.

V. RELATED WORK

Agentic workflows and system implications. LLM-MAS systems have evolved from ReAct-style loops [41] to plan-based decompositions [35] and mixtures of specialized experts [34, 12], introducing structured heterogeneity and dependency amplification that place unique demands on serving infrastructure. Parrot [17] exposes semantic variables to optimize dataflow and Hermes [18] models demand uncertainty via probabilistic graphs and Gittins-index policies; Maestro further introduces a tool-intent-aware cost estimator driving SRTF to attack dependency-induced head-of-line blocking.

TABLE VIII
IMPACT OF NODE FITNESS SCORING ON INTERACTIVE QUEUEING DELAY.

Arrival rate λ	Baseline (s)	BinPack Only (s)	Maestro-Aff (s)
0.5	0.12	0.05	**0.03**
1.0	1.79	1.43	**1.23**
2.0	36.99	33.67	**30.77**

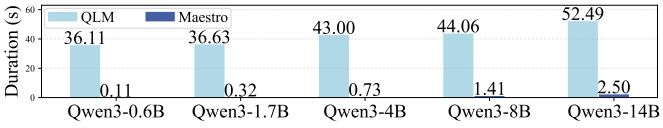

Fig. 10. Model activation latency (0.6B–14B) on a single A100 40GB GPU.

Efficient serving and resource management. Modern LLM serving engines, such as vLLM [16] and SGLang [50], improve throughput through paged attention and structured execution. MuxServe [9] and Castor [20] further increase GPU utilization through spatial-temporal multiplexing. Others target architecture-specific optimizations: UnifiedServe [48] optimizes multi-stage MLLM inference via GPU resource sharing, FinDEP [24] improves MoE inference via fine-grained expert scheduling, and Cauchy [46] adaptively places prefill and decode on heterogeneous GPUs. Similar orchestration principles appear in adjacent domains, including OREO [23] for O-RAN and Mina [7] for in-network aggregation. Serverless and elastic frameworks such as Torpor [43], SMLT [1], Espresso [51], and KAIOps [36] address high concurrency and heterogeneous GPU resources, while QLM [25] and SELA [33] manage model switching to meet SLOs. However, these systems largely treat requests as independent. In contrast, Maestro incorporates agent-level context and workflow dependencies into memory coordination and scheduling, enabling efficient multi-agent orchestration.

Scheduling and performance prediction. Previous work predicts LLM output length using bucketed models (S^3 [15]), semantic embeddings (Magnus [6]), or neural regressors [26], but typically ignores the structured context of multi-turn, role-driven agent workflows. In GPU cluster scheduling, Kale [19] leverages traffic forecasting for elastic autoscaling, Cuckoo [47] jointly optimizes deadline satisfaction and utilization via spatio-temporal packing, A-SRPT [21] couples prediction with SRPT scheduling, and KAIR [40] employs causal inference to diagnose training stragglers. Maestro extends prediction-driven scheduling to the multi-agent setting by combining a two-phase tool-intent predictor with fitness-based cross-cluster routing and workflow-aware remaining-time queueing.

VI. CONCLUSION

We presented Maestro, a workload-aware scheduling system for LLM-MAS. Rather than treating LLM requests as opaque, Maestro characterizes the workload through agent-level dimensions—role and workflow position, tool-invocation intent, predicted per-stage output length and KV footprint, and per-cluster model readiness—and drives hierarchical weight residency, elastic KV provisioning, fitness-based dispatch, and boundary-preemptive SRTF in a single control loop. Across prototype experiments and trace-driven simulations, Maestro reduces KV-reservation HBM by 67.2% and improves high-contention SLO attainment over EDF by 23.6 percentage points.

Limitations and future work. Maestro complements—rather than replaces—kernel-/model-level optimizations (speculative decoding, KV compression, adapter-based serving, disaggregated prefill/decode, iteration-level schedulers); these can be folded in by updating per-model microbenchmarks and readiness signals. Our current prototype focuses on homogeneous A100 nodes and two service classes; production deployments with heterogeneous accelerators and multiple tenants require hardware-aware placement constraints, quota enforcement, and stronger fairness guarantees. Finally, although tool latency is recorded in workflow profiles when available, jointly scheduling external tool execution with LLM inference remains an important direction for tool-heavy agent systems.

ACKNOWLEDGMENT

This work was supported in part by National Key R&D Program of China (Grant No. 2024YFB4505604), in part by the National Natural Science Foundation of China (Grant No. 62402024), in part by the Fundamental Research Funds for the Central Universities.

REFERENCES

[1] Ahsan Ali et al. “Enabling scalable and adaptive machine learning training via serverless computing on public cloud”. *Performance Evaluation 2025*.

[2] Alibaba Cloud. *Internet Access Performance*. Alibaba Cloud documentation.

[3] Yuntao Bai et al. *Training a Helpful and Harmless Assistant with Reinforcement Learning from Human Feedback*. arXiv:2204.05862. 2022.

[4] Yushi Bai et al. “LongBench v2: Towards Deeper Understanding and Reasoning on Realistic Long-context Multitasks”. *ACL 2025*.

[5] Weize Chen et al. “AgentVerse: Facilitating Multi-Agent Collaboration and Exploring Emergent Behaviors”. *ICLR 2024*.

[6] Ke Cheng et al. “Enabling efficient batch serving for LMaaS via generation length prediction”. *ICWS 2024*.

[7] Shichen Dong et al. “Mina: Fine-Grained In-network Aggregation Resource Scheduling for Machine Learning Service”. *IEEE INFOCOM 2025*.

[8] Yilun Du et al. “Improving factuality and reasoning in language models through multiagent debate”. *ICML 2024*.

[9] Jiangfei Duan et al. “MuxServe: flexible spatial-temporal multiplexing for multiple LLM serving”. *ICML 2024*.

[10] Arpan Gujarati et al. “Serving DNNs like Clockwork: Performance Predictability from the Bottom Up”. *USENIX OSDI 2020*.

[11] Daya Guo et al. *DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning*. arXiv:2501.12948. 2025.

[12] Taicheng Guo et al. “Large language model based multi-agents: a survey of progress and challenges”. *IJCAI 2024*.

[13] Dan Hendrycks et al. “Measuring Mathematical Problem Solving With the MATH Dataset”. *NeurIPS 2021*.

[14] Sirui Hong et al. “MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework”. *ICLR 2024*.

[15] Yunho Jin et al. “S3: Increasing GPU Utilization during Generative Inference for Higher Throughput”. *NeurIPS 2023*.

[16] Woosuk Kwon et al. “Efficient memory management for large language model serving with pagedattention”. *SOSP 2023*.

[17] Chaofan Lin et al. “Parrot: Efficient serving of LLM-based applications with semantic variable”. *OSDI 2024*.

[18] Yifei Liu et al. *Efficient Serving of LLM Applications with Probabilistic Demand Modeling*. arXiv:2506.14851. 2025.

[19] Ziyang Liu et al. “Kale: Elastic GPU Scheduling for Online DL Model Training”. *ACM SoCC 2024*.

[20] Yizhou Luo et al. “Castor: Optimizing Deep Learning Job Scheduling in Multi-Tenant GPU Clusters via Intelligent Colocation”.

[21] Ziyue Luo et al. “Prediction-assisted online distributed deep learning workload scheduling in GPU clusters”. *IEEE INFOCOM 2025*.

[22] Ziming Mao et al. “SkyServe: Serving AI Models across Regions and Clouds with Spot Instances”. *EuroSys 2025*.

[23] Federico Mungari et al. “O-RAN Intelligence Orchestration Framework for Quality-Driven xApp Deployment and Sharing”. *IEEE Transactions on Mobile Computing 2025*.

[24] Xinglin Pan et al. *Efficient MoE Inference with Fine-Grained Scheduling of Disaggregated Expert Parallelism*. arXiv:2512.21487. 2025.

[25] Archit Patke et al. “Queue management for slo-oriented large language model serving”. *ACM SoCC 2024*.

[26] Haoran Qiu et al. “Efficient Interactive LLM Serving with Proxy Model-based Sequence Length Prediction”. *ASPLOS Cloud Intelligence/AIOps Workshop 2024*.

[27] Pranav Rajpurkar et al. “SQuAD: 100,000+ Questions for Machine Comprehension of Text”. *EMNLP 2016*.

[28] David Rein et al. *GPQA: A Graduate-Level Google-Proof Q&A Benchmark*. arXiv:2311.12022. 2023.

[29] Ying Sheng et al. “FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU”. *ICML 2023*.

[30] Pavan Subhash. *IBM HR Analytics Employee Attrition & Performance*. Kaggle dataset. 2017.

[31] The AIBrix Team et al. *AIBrix: Towards Scalable, Cost-Effective Large Language Model Inference Infrastructure*. arXiv:2504.03648. 2025.

[32] Hugo Touvron et al. *LLaMA: Open and Efficient Foundation Language Models*. arXiv:2302.13971. 2023.

[33] Shreshth Tuli et al. “SELA: Smart Edge LLM Agent to Optimize Response Trade-offs of AI Assistants”. *PACM IMWUT 2025*.

[34] Junlin Wang et al. *Mixture-of-agents enhances large language model capabilities*. arXiv:2406.04692. 2024.

[35] Lei Wang et al. “Plan-and-Solve Prompting: Improving Zero-Shot Chain-of-Thought Reasoning by Large Language Models”. *ACL 2023*.

[36] Zeying Wang et al. “KAIOps: A Platform Solution of End-to-End Multi-Modal AIOps for AI Training at Scale”. *IEEE/ACM ASE 2025*.

[37] Bingyang Wu et al. *Fast Distributed Inference Serving for Large Language Models*. arXiv:2305.05920. 2024.

[38] Qingyun Wu et al. *AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation*. arXiv:2308.08155. 2023.

[39] An Yang et al. *Qwen3 Technical Report*. arXiv:2505.09388. 2025.

[40] Yitang Yang et al. “KAIR: A Statistical and Causal Approach to Pinpointing Stragglers in Distributed Model Training”. *IEEE/ACM ASE 2025*.

[41] Shunyu Yao et al. “ReAct: Synergizing reasoning and acting in language models”. *ICLR 2023*.

[42] Gyeong-In Yu et al. “Orca: A Distributed Serving System for Transformer-Based Generative Models”. *USENIX OSDI 2022*.

[43] Minchen Yu et al. “Torpor: GPU-enabled serverless computing for low-latency, resource-efficient inference”. *USENIX ATC 2025*.

[44] Shan Yu et al. “Prism: Unleashing GPU Sharing for Cost-Efficient Multi-LLM Serving”. 2026.

[45] Yijiong Yu et al. *OpenCSG Chinese Corpus: A Series of High-quality Chinese Datasets for LLM Training*. arXiv:2501.08197. 2025.

[46] Yihui Zhang et al. “Cauchy: A Cost-Efficient LLM Serving System through Adaptive Heterogeneous Deployment”. *ACM SoCC 2025*.

[47] Yuzhang Zhang et al. “Cuckoo: Deadline-Aware Job Packing on Heterogeneous GPUs for DL Model Training”. *ACM SoCC 2025*.

[48] Lingxiao Zhao et al. *Enabling Disaggregated Multi-Stage MLLM Inference via GPU-Internal Scheduling and Resource Sharing*. arXiv:2512.17574. 2025.

[49] Lianmin Zheng et al. “Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena”. *NeurIPS 2023*.

[50] Lianmin Zheng et al. “SGLang: efficient execution of structured language model programs”. *NeurIPS 2024*.

[51] Qiannan Zhou et al. “Espresso: Cost-Efficient Large Model Training by Exploiting GPU Heterogeneity in the Cloud”. *IEEE INFOCOM 2025*.