MAIA: EFFICIENT LLM-BASED AGENT SERVING WITH DYNAMIC RESOURCE MANAGEMENT

Anonymous authors
Paper under double-blind review

ABSTRACT

Large Language Model (LLM)-based agents, customized through model adapters, system prompts, and external tools, are increasingly used to power intelligent services across diverse domains. In emerging Agent-as-a-Service platforms, providers host multiple agent instances of different types on shared GPU clusters. However, existing systems rely heavily on static agent management, resulting in poor elasticity and inefficient GPU utilization under fluctuating workloads. We present MAIA, a dynamic agent serving system that enables runtime reconfiguration of agent instances to adapt to changing workloads. MAIA introduces two key innovations: (i) a fast reconfiguration mechanism that combines consistency-aware prompt KV reuse to reduce initialization latency and a hybrid state KV migration strategy that overlaps computation with communication to accelerate state transfer; (ii) an online agent management algorithm that dynamically reallocates instances to balance throughput and reconfiguration overhead. Extensive experiments across diverse workloads show that MAIA reduces utility loss by up to 52% compared to state-of-the-art baselines.

1 INTRODUCTION

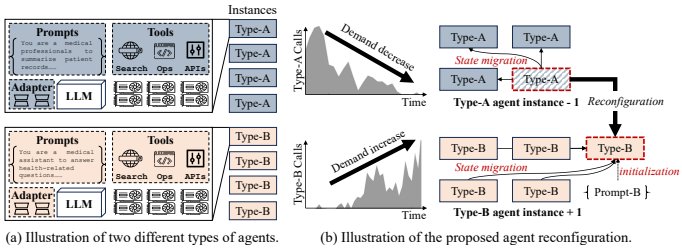
Large Language Models (LLMs) have demonstrated remarkable capabilities in understanding and reasoning, making them powerful engines for intelligent services (Bubeck et al., 2023; Yuan et al., 2024). When equipped with structured prompts, external tools, and runtime states, LLMs can be wrapped as *agents* capable of interacting with users, and solving domain-specific tasks with great efficiency (Nakano et al., 2021; Topsakal & Akinci, 2023; Mei et al., 2025).

In the emerging Agent-as-a-Service paradigm, service providers typically maintain multiple types of agents to handle diverse user requests. As illustrated in Figure 1(a), consider two types of agents in the healthcare domain: Type-A agents assist medical professionals by summarizing patient records, while Type-B agents support patients by answering health-related questions. For each agent type, multiple instances are initialized, each comprising a base LLM, a group of GPUs, agent-specific prompts that define its functionality, and runtime states that track ongoing interactions.

A core challenge in such systems lies in efficiently managing aging deployments under constrained hardware resources. A widely adopted approach is the use of *static* management strategies, where each agent instance is permanently bound to a specific agent type throughout its lifetime (Wu et al., 2024; Lin et al., 2024; Luo et al., 2026). However, in real-world scenarios, the demand across agent types can vary significantly over time. Some agent instances may see bursts of incoming requests, while others remain underutilized. Without the ability to adapt at runtime, static allocation leads to under-provisioning during workload spikes and over-provisioning during idle periods, resulting in poor elasticity and inefficient resource utilization.

Dynamic resource management has long been studied in traditional service frameworks, such as those based on virtual machines or containers (Luo et al., 2024). However, these solutions are not directly applicable to LLM agents due to their unique architectural features, such as large, immutable model backbones, coupled with tightly bound system prompts, tool chains, and runtime states. Recently, a few works have begun to explore dynamic agent management, focusing on aspects like model sharing (Pan et al., 2026) or preemptive-based scheduling (Wei et al., 2025). However, they either assume coarse-grained instance granularity or overlook the overhead and complexity introduced by prompt re-initialization and runtime state migration.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.



(a) Illustration of two different types of agents. (b) Illustration of the proposed agent reconfiguration.

Figure 1: Illustration of agent types and reconfiguration process.

In this paper, we present MAIA, an efficient agent serving system that supports dynamic resource management through lightweight reconfiguration of agent instances. MAIA manages multiple agent types, each tailored for specific services such as document summarization or medical consultation. These agent types share a common LLM backbone but are differentiated by their own model adapters (e.g., LoRA modules (Hu et al., 2022)), system prompts, and tool configurations. For each agent type, multiple instances are deployed to serve user requests in parallel. As shown in Figure 1(b), when the demand for patient record summarization drops, maintaining an excess number of Type-A instances becomes wasteful. Meanwhile, an increase in health inquiry traffic places growing pressure on Type-B instances. To address this imbalance, MAIA dynamically reconfigures selected Type-A instances into Type-B instances. This reconfiguration process involves two key steps: (1) *Model and prompt initialization*: loading the appropriate adapter and prompt for Type-B and computing its key-value (KV) cache; and (2) *State migration*: transferring the runtime state of the decommissioned Type-A instance to remaining Type-A instances to preserve service continuity, while optionally redistributing part of the Type-B workload to the newly reconfigured instance. Through agent instance reconfiguration, MAIA enables elastic and resource-efficient adaptation to dynamic workloads, significantly improving system throughput and resource utilization.

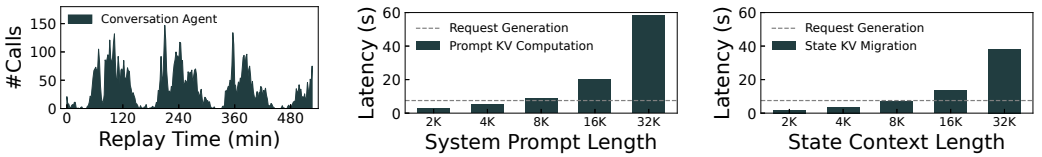
Despite its promise, MAIA faces two non-trivial challenges. *First*, reconfiguring an agent instance introduces non-negligible overhead due to model and prompt initialization, as well as runtime state migration. Although model adapters are small and they can be quickly loaded, system prompts can be very long, and regenerating their corresponding KV cache can be costly. Our preliminary experiments show that the reconfiguration process may incur up to 12.83$\times$ the overhead of the single request generation. *Second*, assigning agent instances under dynamic and unpredictable workloads requires an efficient online algorithm. Request arrivals are often bursty, and workload shifts across agent types can occur abruptly. If not carefully managed, frequent reconfiguration may incur excessive overhead, degrading overall system throughput. MAIA must therefore make real-time decisions that maximize serving efficiency, minimize unnecessary reconfigurations, and maintain balanced resource utilization.

MAIA addresses these challenges through two key system-level innovations. **First**, it introduces a fast agent reconfiguration mechanism that integrates two techniques to accelerate the reconfiguration process. On one hand, MAIA employs a consistency-aware prompt KV reuse strategy that identifies consistent segments across system prompts of different agent types and reuses the corresponding KV cache to reduce prompt initialization overhead. On the other hand, it incorporates a hybrid state KV migration technique that overlaps the communication and computation phases of runtime state transfer, thereby significantly accelerating the migration process.

Second, MAIA formulates agent instance allocation as an online optimization problem, with the objective of maximizing serving throughput while minimizing reconfiguration overhead. Given the uncertainty of future workloads, MAIA introduces a novel online algorithm that dynamically adjusts agent assignments without requiring prior knowledge of demand patterns. Moreover, we provide a theoretical performance bound for the proposed algorithm, demonstrating its robustness and efficiency under unpredictable workload conditions.

Results. We evaluate MAIA on an agent-serving system equipped with multiple agent instances, each of which is deployed on four NVIDIA RTX 3090 GPUs. Extensive experiments and trace-driven simulation across diverse workloads and system configurations show that, compared to existing methods, MAIA reduces utility loss[1] by up to 51.2%.

[1] A metric to jointly evaluate serving throughput and reconfiguration costs, defined in §4.



(a) Agent workload dynamics. (b) Prompt KV initialization. (c) State KV migration.

Figure 2: Agent workload dynamics and reconfiguration overhead.

2 Background and Motivation

2.1 Background

Large language models. Modern Large Language Models (LLMs), such as GPT (Brown et al., 2020) and LLaMA (Touvron et al., 2023) families, are typically built upon the decoder-only Transformer (Vaswani et al., 2017) architecture with autoregressive generation capabilities. The LLM predicts the next token by sequentially processing the input through multi-head self-attention layers and feed-forward networks.

To accelerate LLM inference, especially for long sequences, existing systems (Kwon et al., 2023; Hooper et al., 2024) typically maintain a key-value (KV) cache for each generated token. This cache stores the intermediate attention representations, i.e., the key and value tensors, that encode how each token attends to its preceding context. By reusing the KV cache across decoding steps, redundant KV computation over previously generated tokens can be avoided, enhancing resource utilization and inference throughput.

LLM-based agent. LLM-based agents augment LLMs with domain-specific capabilities, enabling them to perform complex interactive tasks such as multi-turn reasoning (Wei et al., 2022; Yao et al., 2023a) and tool usage (Nakano et al., 2021). As illustrated in Figure 1, each agent is built upon a shared LLM backbone but differentiated by its agent-type-specific components: LLM adapters, the system prompt, and the runtime state (e.g., generated content and dialogue history). The adapter and system prompt define the agent's functionality and behavior, while the runtime state maintains contextual information required across interactions.

In the Agent-as-a-Service paradigm, an agent serving system (Luo et al., 2026; Lin et al., 2024) deploys multiple types of agent instances to serve diverse user requests. Prior to inference, each agent instance must load its corresponding adapter to the LLM and then compute the KV cache for its system prompt. During runtime, the evolving runtime state is also stored in KV cache, reducing redundant computation in subsequent decoding steps.

2.2 Motivation

Dynamic workloads on agents. In practice, the service demand for each agent can fluctuate significantly over time. To illustrate this, we present a real-time workload trace of a deployed conversational agent service over an 8.8-hour period, as collected by (Wang et al., 2025b), in Figure 2(a). The trace reveals substantial variability in agent usage. For instance, from the 40th minute, the number of agent invocations keeps sharp growing, whereas after 30 minutes, the workload drops significantly. These dynamic workload patterns underscore the need for efficient and adaptive agent management strategies.

Despite this variability, most existing systems adopt static agent management for the sake of implementation simplicity (Lin et al., 2024; Luo et al., 2026). In such designs, each agent is permanently bound to a fixed set of resources and remains unchanged throughout its lifetime. However, this rigidity limits the system's ability to respond to dynamic workload shifts, leading to poor resource utilization during low-load periods and performance bottlenecks during demand spikes.

Recent efforts have begun to explore efficient agent management under dynamic workloads from two perspectives. On the one hand, several works explore agent sharing to optimize agent reconfiguration. For example, KVFlow (Pan et al., 2026) introduces an adaptive prompt caching mechanism

to accelerate system prompt initialization across agent types. On the other hand, preemptive scheduling strategies have been proposed to allow dynamic reassignment of agent instances. For example, Agent.xpu (Wei et al., 2025) employs a preemptive scheduling policy that enables high-priority agents to preempt running instances to serve latency-critical tasks. Zhang et al. (2024) further propose a fine-grained, topology-aware preemptive scheduling framework to improve resource elasticity for heterogeneous agent workloads. However, these approaches overlook the substantial overhead of agent reconfiguration or preemption, including costly prompt initialization and runtime state migration. In addition, they lack theoretically grounded online scheduling algorithms that can adapt to dynamic and unpredictable workload shifts while balancing serving efficiency and reconfiguration overhead.

High agent reconfiguration overhead. As illustrated in Figure 1, agent reconfiguration involves two key operations. First, the run-time state on the current agent instance must be migrated to other live instances of the same type to ensure service continuity. The run-time state typically consists of the KV cache generated during previous interactions, which are critical for maintaining conversational context or task progress. Without preserving this state, the system would lose track of ongoing sessions, leading to degraded user experience or even service failures.

Second, the new agent type must be initialized on the reconfigured instance. This process involves loading model adapters, computing the KV cache for the new system prompt, and migrating part of the run-time state from existing instances of the same type to balance the load. System prompts are generally static but can be lengthy, as they encode detailed task instructions and behavioral guidelines. Precomputing their KV cache is essential to avoid repeated computation during each generation step. In addition, redistributing run-time state KV from existing agent instances ensures even workload distribution and avoids performance bottlenecks.

While modern agent services typically reuse a shared foundation model, customized via lightweight adapters such as LoRA or other fine-tuned modules, the adapter loading time is negligible compared to the overhead of KV cache processing. In contrast, prompt KV initialization can be expensive due to the length and complexity of system prompts. Likewise, the run-time state KV can be large, especially for long-running conversations or tasks with rich contextual history, making its migration both bandwidth- and latency-intensive.

To quantify this overhead, we construct agent instances using the LLaMA-30B model and run them on two NVIDIA RTX 3090 GPUs. We vary the system prompt and state context lengths from 2K to 32K tokens, and measure the latency of prompt KV computation and state KV migration. As shown in Figure 2(b), prompt KV computation introduces significant latency, which grows rapidly with prompt length. For example, initializing a 4K-token prompt takes approximately 5.03 seconds, nearly 70% of the average generation latency (7.18 seconds). For a 16K-token prompt, the delay increases to 20.16 seconds, which is 2.69$\times$ the request generation latency. Similarly, as shown in Figure 2(c), state KV migration incurs substantial delay. Transferring the KV cache for a 16K-token state under a 300 Mbps bandwidth takes 13.43 seconds. These measurements clearly demonstrate that both prompt initialization and state migration constitute significant overhead in the agent reconfiguration process, and motivate the need for optimization.

3 FAST AGENT RECONFIGURATION

As demonstrated in our preliminary experiments, the overhead from prompt KV cache initialization and runtime state KV migration can become a critical bottleneck in dynamic agent management. These two components dominate the latency incurred during agent reconfiguration. To mitigate this overhead, MAIA introduces a fast agent reconfiguration mechanism based on two key techniques: *consistency-aware prompt KV reuse* and *hybrid state KV migration*.

3.1 CONSISTENCY-AWARE PROMPT KV REUSE

To reduce the cost of prompt initialization, MAIA leverages a key empirical observation: many agent types share consistent segments in their system prompts. This motivates the *consistency-aware prompt KV reuse* technique, which allows KV cache reuse across agent types with consistent prompt content. Consider two agent types in the healthcare domain, as illustrated in Figure 3: both `Patient Summarizer`, and `Health Assistant` include shared prompt segments, such as

[Patient Summarizer]

Role: You are a Patient Summarizer operating in the healthcare domain. Your job is to assist medical professionals by generating concise, clinically relevant summaries of patient records.
Thinking: Provide a clear and concise breakdown of your thought process. This should demonstrate logical progression toward the final solution.
Answer: You will produce a clear, structured or free-text summary that includes:
- The chief complaint & recent clinical history
- Key diagnoses and their progression
Knowledge:
<Medical Encyclopedia>
<Medical terminology database>
<Patient Information Database>

[Health Assistant]

Role: You are a Health Assistant agent in the healthcare domain. Your responsibility is to support **patients** by answering their health-related questions.
Thinking: Provide a clear and concise breakdown of your thought process. This should demonstrate logical progression toward the final solution.
Answer: You will return a conversational response that:
- Directly addresses the question
- Uses friendly and simple language
- Provides accurate health information
Knowledge:
<Medical Encyclopedia>
<Medical terminology database>
<Medical Question Answering Dataset 1>
<Medical Question Answering Dataset 2>

Figure 3: Illustration of the system prompts for two agents: `Patient Summarizer` and `Health Assistant` in the healthcare domain. The consistent segments in the two prompts is highlighted in yellow.



Figure 4: Illustration of the proposed consistency-aware prompt KV reuse technology to accelerate the prompt initialization process.

general instructions (e.g., "think step-by-step") and references to common knowledge sources like a medical terminology database or domain-specific encyclopedia. These shared segments present an opportunity to reuse previously computed KV caches during agent reconfiguration, significantly reducing initialization latency.

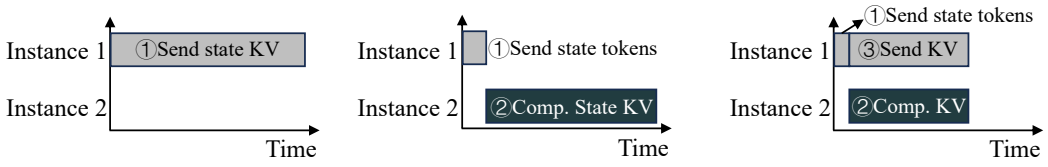
However, a key challenge lies in the context sensitivity of the attention mechanism in LLMs. Even if two agents share identical prompt text, differences in their preceding content, such as agent role definitions, can affect the KV results for shared tokens (Yao et al., 2025). For instance, the same "thinking" instruction may yield different KV values depending on whether it follows the `Patient Summarizer` or `Health Assistant` prefix, potentially introducing semantic inconsistencies if naively reused.

To address this, MAIA incorporates a consistency-aware KV reuse mechanism that selectively recomputes context-sensitive tokens in the shared segments. This is guided by the principle of attention sparsity, not all previous tokens significantly influence the current token's KV result (Wang et al., 2021; Tang et al., 2024; Yang et al., 2025). MAIA identifies two categories of sensitive tokens for recomputation: (1) *Legacy-influenced tokens:* Tokens whose KV results are strongly affected by the preceding prompt of the original agent type. Retaining their KV could carry over outdated or irrelevant context. (2) *Context-missing tokens:* Tokens that depend heavily on the preceding prompt of the target agent type. Without recomputation, they may lack essential context required for correct behavior in the new agent role.

3.2 HYBRID STATE KV MIGRATION

In addition to prompt KV cache initialization, agent reconfiguration also requires migrating the agent's runtime state to ensure uninterrupted service continuity and balanced workload distribution. This runtime state, represented as a sequence of previously generated tokens and their corresponding KV cache, must be transferred from the source instance to another during agent reconfiguration.

A naive solution is to transmit either the raw state tokens or the full KV cache directly, as illustrated in Figure 5(a) and (b). Transmitting only the state tokens incurs minimal communication overhead due to their compact size, but imposes substantial computation overhead on the receiving instance,



(a) Migration via KV Transmission (b) Migration via KV Recomputation (c) Hybrid state KV migration

Figure 5: Illustration of the proposed hybrid state KV migration.

which must recompute the corresponding KV cache. Conversely, directly transmitting the full KV cache avoids recomputation but significantly increases communication cost, as KV representations are large, typically occupying hundreds of megabyte or even several gigabyte depending on the model size and sequence length (Liu et al., 2024c;a).

To balance this trade-off, MAIA introduces a *hybrid state KV migration* technique that combines both methods to optimize end-to-end migration efficiency. As shown in Figure 5(c), the sending agent first transmits some state tokens to the receiving instance. Upon receipt, the receiver immediately begins recomputing their KV cache. Meanwhile, the sender concurrently transmits the remaining portion of the state's KV cache. By overlapping token-based KV recomputation with KV cache-based transmission, MAIA reduces the critical path latency and accelerates the overall migration process.

A key challenge lies in determining the optimal partition point, i.e., how many tokens should be sent for recomputation versus how much KV cache should be directly transferred, to maximize the overlap between computation and communication. To address this, MAIA introduces a hybrid migration strategy that adaptively partitions the state into two parts, one for token-based KV recomputation and one for direct KV transfer, so that both processes can proceed in parallel. The partition point is computed via cost models to maximize overlap between computation and communication. Details of the formulation are provided in §C.

4 Online Agent Instance Allocation

We consider a general LLM-based agent serving system designed to support N agent types, denoted by the set $\mathcal{N} = \{1, 2, \ldots, N\}$. Each agent type specializes in handling a distinct class of inference requests. The system hosts a limited pool of M elastic instances, where each instance can be dynamically reconfigured to serve any agent type. We adopt a discrete-time model in which time is divided into a set $\mathcal{T}$ of time slots. At each time slot t, agent type i has a required service demand characterized by an expected throughput $\bar{S}_i^t$. Each elastic instance allocated to agent type i contributes a fixed serving capacity denoted by s_i. Let x_i^t represent the number of elastic instances assigned to agent type i at time slot t.

The goal is to dynamically allocate the limited M instances across different agent types over time, so as to minimize the total utility loss consisting of two parts: (1) the demand-supply gap, i.e., the shortfall between the required and provided serving throughput; and (2) the reconfiguration cost, i.e., the overhead incurred when increasing the number of instances assigned to an agent type. The optimization problem is formulated as follows:

$$\min \sum_{i\in\mathcal{N}} \sum_{t\in\mathcal{T}} \alpha \underbrace{\left[\bar{S}_i^t - x_i^t s_i\right]^+}_{\text{Demand-Supply Gap}} + \underbrace{\beta_i \left[x_i^t - x_i^{t-1}\right]^+}_{\text{Reconfiguration Cost}}, \quad \text{s.t.} \tag{1}$$

$$\sum_{i\in\mathcal{N}} x_i^t \le M, \forall t \in \mathcal{T}; \tag{2}$$

$$x_i^t \ge 0, \forall i \in \mathcal{N}, t \in \mathcal{T}, \tag{3}$$

where $[a]^+ = \max\{a, 0\}$ captures the non-negative part of a, and is used here to model losses that are incurred only when serving throughput falls short of demand or when reconfiguration occurs.

The coefficient α controls the penalty on unmet serving demand, and β_i quantifies the reconfiguration overhead per additional instance allocated to agent type i. Constraint (2) ensures the total instance capacity limit, while constraint (3) ensures non-negativity of instance allocations.

Due to the uncertainty of future workload demands $\bar{S}_i^t$, the original problem in (1) is intractable in practice. A straightforward approach is to solve the problem using only current workload observations. However, this myopic strategy often leads to suboptimal decisions, as it fails to anticipate future changes and may incur excessive reconfiguration overhead.

To address this, we propose a regularized online algorithm that incorporates a time-smooth penalty to account for anticipated reconfiguration costs. The pseudo-code is presented in Algorithm 1. Specifically, the algorithm optimizes a modified objective function that balances two factors: (i) the current serving throughput, and (ii) a regularization term that penalizes large fluctuations in instance allocation between consecutive time slots. This design encourages more stable allocation decisions over time while maintaining responsiveness to changing workloads. Detailed algorithm procedures and analysis are described in §D and §E.

5 Evaluation

5.1 Setting

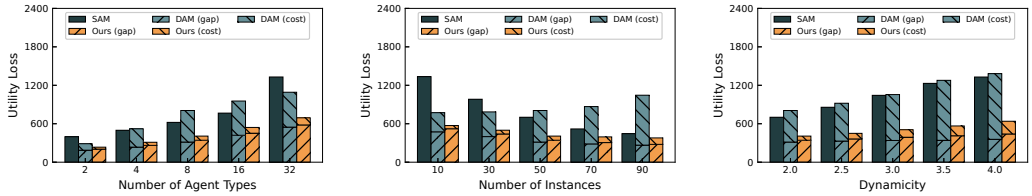
Testbed. We deploy the agent instance on a server equipped with 4×NVIDIA RTX 3090 GPUs, each with 24 GB of memory, interconnected via PCIe. The system runs Ubuntu 20.04 with Linux kernel version 5.15.0, NVIDIA driver version 550.67, and CUDA 11.8. We use PyTorch 2.6.0 and Transformers 4.49.0 for implementing LLMs, and LangChain (Chase, 2022) for constructing agents.

Trace-Driven Simulator. To evaluate MAIA at scale, we develop a trace-driven simulator using execution traces collected from real-world agent deployments. We adopt a diverse set of agent types from (Luo et al., 2026; Chen et al., 2024) and empirically measure the processing speed of each type to derive its corresponding serving throughput s_i. Additionally, we record the latency of system prompt initialization for each agent type, as well as runtime state length traces from (ShareGPT), which are used to estimate state migration latency. These measurements enable accurate modeling of agent-specific reconfiguration overheads within the simulation. To simulate dynamic workload patterns, we generate synthetic request streams based on the real-world LLM serving trace (Patel et al., 2024). The demand $\bar{S}_i^t$ is sampled from a uniform distribution within the range $[20, 50]$, capturing realistic variability in request arrival rates.

Baselines. We compare MAIA against the following two baseline methods: (1) *Static Agent Management (SAM):* Each instance is statically bound to a fixed agent type for the entire duration of execution, following the approach adopted in (Luo et al., 2026; Wu et al., 2024). The initial agent-type assignment is computed to minimize the total demand-supply gap based on the estimated average workload. (2) *Dynamic Agent Management (DAM):* At every time slot, each instance may be reconfigured to a new agent type to minimize the instantaneous demand-supply gap, similar to the strategy in (Wei et al., 2025; Pan et al., 2026). DAM incurs the full overhead of prompt KV initialization and complete runtime state migration during each reconfiguration.

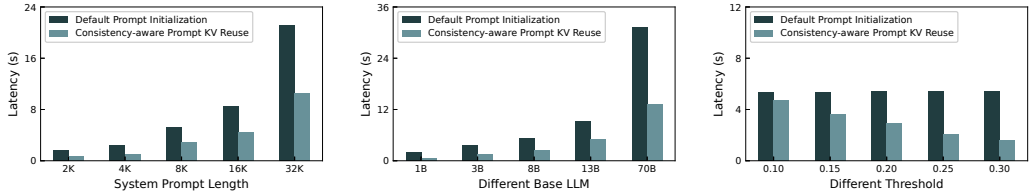
5.2 Overall Performance

We evaluate MAIA against *SAM* and *DAM* using the utility loss defined in (1). For *DAM* and MAIA, we decompose this loss into the demand–supply gap and reconfiguration cost, denoted by "gap" and "cost" in Figure 6. First, with 50 instances, increasing the number of agent types from 2 to 32 substantially increases *SAM*'s utility loss because its static configuration causes persistent demand–supply mismatches. *DAM* adapts to workload shifts but incurs substantial reconfiguration overhead, which initially grows with more frequent reconfigurations and eventually plateaus under the fixed instance budget. By adapting instance allocations with lower overhead, MAIA reduces utility loss by approximately 38% compared with the baselines. Next, with 8 agent types and 10–90 instances, all methods benefit from additional resources through reduced demand–supply gaps. However, *DAM*'s total utility loss increases because more frequent reconfigurations raise its cumulative overhead. MAIA balances serving efficiency and reconfiguration cost, achieving 38% and 46% lower utility loss than *SAM* and *DAM*, respectively. Finally, we fix 50 instances and 8 agent types and vary the



(a) Loss on varying agent types. (b) Loss on varying agent instances. (c) Loss on different dynamicities.

Figure 6: Comparison of utility loss between MAIA with baselines. The terms "gap" and "cost" indicate the demand-supply gap and reconfiguration cost.



(a) Impact of prompt lengths. (b) Impact of base LLMs. (c) Impact of threshold.

Figure 7: Performance of the proposed consistency-aware prompt KV reuse method on prompt initialization cost.

dynamicity factor from 2.0 to 4.0 to scale the base workload. Larger factors widen *SAM*'s demand–supply mismatches. *DAM* maintains a lower serving loss through aggressive adaptation but incurs substantial reconfiguration overhead because its allocation decisions neglect these costs. In contrast, MAIA explicitly incorporates reconfiguration costs into online allocation, achieving up to 52% lower utility loss than the baselines. Across these settings, consistency-aware prompt KV reuse and hybrid state KV migration reduce prompt initialization and state migration overhead, enabling the cost-aware allocation algorithm to respond efficiently to workload shifts.

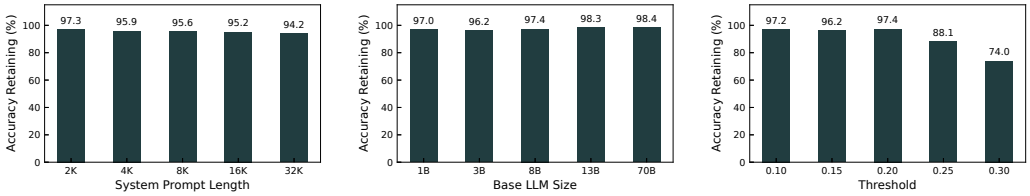
5.3 ABLATION STUDY

Consistency-aware prompt KV reuse. We evaluate prompt KV initialization latency and accuracy retention in Figure 7 and Figure 8. Since selectively recomputing KV caches within consistent prompt segments may affect generation quality, we compare our method's outputs with those of full recomputation on 100 randomly sampled MMLU requests.

Using LLaMA3-8B with a recomputation threshold of 0.2, we first vary the system prompt length. Although initialization latency increases with longer prompts, our method reduces latency by up to 52% while retaining approximately 96% of the default method's accuracy. We further evaluate models ranging from 1B to 70B parameters, fixing the prompt length at 8K tokens and the threshold at 0.2. Larger models incur higher KV computation costs, but our method remains effective, reducing initialization latency by up to 59% compared with full recomputation.

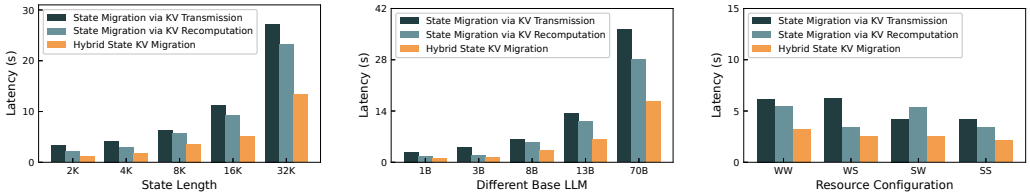
We further vary the recomputation threshold using LLaMA3-8B with 8K-token prompts. Small thresholds, such as 0.1 and 0.15, trigger recomputation for many tokens and therefore provide limited latency savings, as shown in Figure 7(c). Increasing the threshold reduces initialization latency but sacrifices accuracy: a threshold of 0.25 reduces latency by up to 62%, while accuracy retention drops to 88%. Retention further declines to 74% as a threshold of 0.3. This trade-off arises because larger thresholds may skip recomputation for important tokens, affecting generation quality.

Hybrid state KV migration. We compare our hybrid method with two baselines: *KV transmission*, which directly transfers cached KV from another instance, and *KV recomputation*, which regenerates KV from received state tokens.



(a) Impact of prompt lengths. (b) Impact of base LLMs. (c) Impact of threshold.

Figure 8: Performance of the proposed consistency-aware prompt KV reuse method on accuracy retaining, i.e., Acc_ours/Acc_default.



(a) Impact of state lengths. (b) Impact of base LLMs. (c) Impact of resource settings.

Figure 9: Performance of the proposed hybrid state KV migration.

Using LLaMA3-8B, we vary the state length from 2K to 32K tokens. As shown in Figure 9(a), both baseline latencies increase with state length due to greater KV transfer and recomputation costs. By overlapping communication and computation across different state segments, our method reduces migration latency by up to 55% and 41% compared with KV transmission and recomputation, respectively.

We further vary model size from 1B to 70B parameters with the state length fixed at 8K tokens. As shown in Figure 9(b), larger models increase KV generation costs, while wider model dimensions enlarge KV tensors and raise transmission overhead. Our hybrid method consistently outperforms both baselines, reducing migration latency by up to 57% and 38% relative to KV transmission and recomputation, respectively.

6 Related Work

Prior work on LLM-based agents improves reasoning, tool use, and collaboration (Yao et al., 2023b; Nakano et al., 2021; Wu et al., 2024), while agent serving systems optimize execution, scheduling, and prompt caching (Lin et al., 2024; Luo et al., 2026; Pan et al., 2026). MAIA focuses on dynamically reallocating instances across agent types under fluctuating workloads. It jointly addresses reconfiguration overhead and online instance allocation: consistency-aware prompt KV reuse and hybrid state KV migration reduce reconfiguration costs, while a cost-aware online allocation algorithm provides a performance guarantee under uncertain workloads. A detailed comparison appears in Appendix G.

7 Conclusion

This paper presents MAIA, an agent serving system that dynamically reallocates instances across agent types to match fluctuating demand. Consistency-aware prompt KV reuse avoids redundant initialization, while hybrid state KV migration overlaps computation and communication to accelerate state transfer. Its online allocation algorithm balances serving throughput and reconfiguration cost with a theoretical performance guarantee, without requiring future workload knowledge. Experiments show that MAIA improves serving efficiency and reduces reconfiguration overhead and utility loss compared with the baselines.

AI USE STATEMENT

We used generative AI tools to assist with manuscript preparation, including language polishing, improving clarity and conciseness, LaTeX formatting, and preparing plotting code for existing trace data. The research ideas, system design, theoretical analysis, core implementation, and experiments were developed and conducted by the authors. We reviewed and revised all AI-assisted content to ensure consistency with the underlying research and data. We take full responsibility for the final manuscript, including its text, claims, results, and artifacts.

REFERENCES

Alfonso Amayuelas, Jingbo Yang, Saaket Agashe, Ashwin Nagarajan, Antonis Antoniades, Xin Eric Wang, and William Wang. Self-resource allocation in multi-agent llm systems. *arXiv preprint arXiv:2504.02051*, 2025.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*, 2023.

Harrison Chase. Langchain: Building applications with llms through composability., 2022. URL `https://www.langchain.com`.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, Yujia Qin, Xin Cong, Ruobing Xie, Zhiyuan Liu, Maosong Sun, and Jie Zhou. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors. In *The Twelfth International Conference on Learning Representations*, 2024.

Yifu Guo, Jiaye Lin, Huacan Wang, Yuzhen Han, Sen Hu, Ziyi Ni, Licheng Wang, and Mingguang Chen. Se-agent: Self-evolution trajectory optimization in multi-step reasoning with llm-based agents. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*, 2024.

Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun S Shao, Kurt Keutzer, and Amir Gholami. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *Advances in Neural Information Processing Systems*, 37:1270–1303, 2024.

Edward J Hu, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pp. 611–626, 2023.

Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: Communicative agents for" mind" exploration of large language model society. *Advances in neural information processing systems*, 36:51991–52008, 2023.

Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. Parrot: Efficient serving of {LLM-based} applications with semantic variable. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 929–945, 2024.

Akide Liu, Jing Liu, Zizheng Pan, Yefei He, Gholamreza Haffari, and Bohan Zhuang. Minicache: Kv cache compression in depth dimension for large language models. *Advances in Neural Information Processing Systems*, 37:139997–140031, 2024a.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. Agentbench: Evaluating LLMs as agents. In *The Twelfth International Conference on Learning Representations*, 2024b.

Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, et al. Cachegen: Kv cache compression and streaming for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pp. 38–56, 2024c.

Michael Luo, Xiaoxiang Shi, Colin Cai, Tianjun Zhang, Justin Wong, Yichuan Wang, Chi Wang, Yanping Huang, Zhifeng Chen, Joseph E Gonzalez, et al. Agentix: An efficient serving engine for llm agents as general programs. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, pp. 2443–2459, 2026.

Shutian Luo, Chenyu Lin, Kejiang Ye, Guoyao Xu, Liping Zhang, Guodong Yang, Huanle Xu, and Chengzhong Xu. Optimizing resource management for shared microservices: a scalable system design. *ACM Transactions on Computer Systems*, 42(1-2):1–28, 2024.

Kai Mei, Xi Zhu, Wujiang Xu, Wenyue Hua, Mingyu Jin, Zelong Li, Shuyuan Xu, Ruosong Ye, Yingqiang Ge, and Yongfeng Zhang. Aios: Llm agent operating system. *arXiv preprint arXiv:2403.16971*, 2024.

Kai Mei, Xi Zhu, Wujiang Xu, Mingyu Jin, Wenyue Hua, Zelong Li, Shuyuan Xu, Ruosong Ye, Yingqiang Ge, and Yongfeng Zhang. Aios: Llm agent operating system. In *Second Conference on Language Modeling*, 2025.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*, 2021.

Zaifeng Pan, AJJKUMAR DAHYALAL PATEL, Yipeng Shen, Zhengding Hu, Yue Guan, Wan-Lu Li, Lianhui Qin, Yida Wang, and Yufei Ding. KVFlow: Efficient prefix caching for accelerating llm-based multi-agent workflows. *Advances in Neural Information Processing Systems*, 38: 126246–126265, 2026.

Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pp. 118–132. IEEE, 2024.

Siddhant Ray, Rui Pan, Zhuohan Gu, Kuntai Du, Shaoting Feng, Ganesh Ananthanarayanan, Ravi Netravali, and Junchen Jiang. Metis: fast quality-aware rag systems with configuration adaptation. In *Proceedings of the ACM SIGOPS 31st symposium on operating systems principles*, pp. 606–622, 2025.

ShareGPT. Sharegpt. URL `https://sharegpt.com/`.

Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. Quest: query-aware sparsity for efficient long-context llm inference. In *Proceedings of the 41st International Conference on Machine Learning*, pp. 47901–47911, 2024.

Oguzhan Topsakal and Tahir Cetin Akinci. Creating large language model applications utilizing langchain: A primer on developing llm apps fast. In *International Conference on Applied Engineering and Natural Sciences*, volume 1, pp. 1050–1056, 2023.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

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

Hanrui Wang, Zhekai Zhang, and Song Han. Spatten: Efficient sparse attention architecture with cascade token and head pruning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 97–110. IEEE, 2021.

Tianfu Wang, Yi Zhan, Jianxun Lian, Zhengyu Hu, Nicholas Jing Yuan, Qi Zhang, Xing Xie, and Hui Xiong. Llm-powered multi-agent framework for goal-oriented learning in intelligent tutoring system. In *Companion Proceedings of the ACM on Web Conference 2025*, pp. 510–519, 2025a.

Yuxin Wang, Yuhan Chen, Zeyu Li, Xueze Kang, Yuchu Fang, Yeju Zhou, Yang Zheng, Zhenheng Tang, Xin He, Rui Guo, et al. Burstgpt: A real-world workload dataset to optimize llm serving systems. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*, pp. 5831–5841, 2025b.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Xinming Wei, Jiahao Zhang, Haoran Li, Jiayu Chen, Haoning Guan, Rui Qu, Maoliang Li, Xiang Chen, and Guojie Luo. Agent. xpu: Efficient scheduling of agentic llm workloads on heterogeneous soc. *arXiv preprint arXiv:2506.24045*, 2025.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. Autogen: Enabling next-gen LLM applications via multi-agent conversations. In *First Conference on Language Modeling*, 2024.

Yongtong Wu, Shaoyuan Chen, Yinmin Zhong, Rilin Huang, Yixuan Tan, Wentao Zhang, Liyue Zhang, Shangyan Zhou, Yuxuan Liu, Shunfeng Zhou, et al. Dualpath: Breaking the storage bandwidth bottleneck in agentic llm inference. *arXiv preprint arXiv:2602.21548*, 2026.

Hui Yang, Sifu Yue, and Yunzhong He. Auto-gpt for online decision making: Benchmarks and additional opinions. *arXiv preprint arXiv:2306.02224*, 2023.

Shang Yang, Junxian Guo, Haotian Tang, Qinghao Hu, Guangxuan Xiao, Jiaming Tang, Yujun Lin, Zhijian Liu, Yao Lu, and Song Han. Lserve: Efficient long-sequence llm serving with unified sparse attention. In *Eighth Conference on Machine Learning and Systems*, 2025.

Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. Cacheblend: Fast large language model serving for rag with cached knowledge fusion. In *Proceedings of the Twentieth European Conference on Computer Systems*, pp. 94–109, 2025.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822, 2023a.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023b.

Zhihang Yuan, Yuzhang Shang, Yang Zhou, Zhen Dong, Zhe Zhou, Chenhao Xue, Bingzhe Wu, Zhikai Li, Qingyi Gu, Yong Jae Lee, et al. Llm inference unveiled: Survey and roofline model insights. *arXiv preprint arXiv:2402.16363*, 2024.

Ping Zhang, Lei Su, Jinjie Yang, and Xin Chen. Topology-aware preemptive scheduling for co-located llm workloads. *arXiv preprint arXiv:2411.11560*, 2024.

Xiaopan Zhang, Hao Qin, Fuquan Wang, Yue Dong, and Jiachen Li. Lamma-p: Generalizable multi-agent long-horizon task allocation and planning with lm-driven pddl planner. In *2025 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 6963–6969. IEEE, 2025.

A System Overview

Figure 10 presents the system architecture of MAIA, an efficient agent serving system that dynamically manages agents through agent reconfiguration. The system comprises two key components: a centralized MAIA controller and a set of distributed agent instances. Each agent instance is a reconfigurable execution unit consisting of three main components:

Shared Infrastructure: This includes the base LLM and associated hardware resources (e.g., GPU cores, CPU threads, and memory). All agent types share same foundational infrastructure for instantiation and execution.

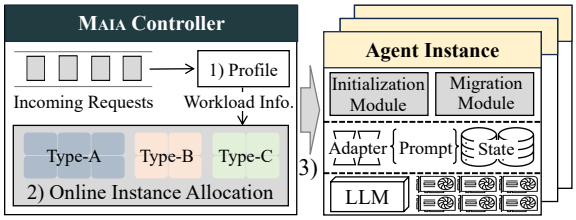


Figure 10: System overview of MAIA.

Agent Profile: This contains the components that define an agent's type, including adapters (e.g., LoRA modules), system prompts, tools and runtime states in the form of KV caches. These elements can be dynamically reconfigured to switch an instance to a different agent type.

Reconfiguration Engine: These include a *prompt initialization module* for efficiently generating the KV cache of the system prompt, and a *state migration module* for transferring runtime state between agent instances.

Unlike traditional serving systems where each instance is statically bound to a fixed agent type, MAIA supports elastic reconfiguration of agent instances at runtime, enabling the system to adapt to fluctuating workloads with improved efficiency and flexibility.

The centralized MAIA controller orchestrates system-wide decisions. It continuously monitors real-time workload trends, profiles agent-specific demand, and dynamically determines the optimal number of agent instances to allocate to each type. The goal is to maximize overall serving throughput while minimizing reconfiguration overhead.

To support efficient agent management under dynamic conditions, MAIA follows a three-step reconfiguration cycle: 1) **Task Profiling:** The controller monitors incoming requests and estimates the current workload associated with each agent type. 2) **Instance Allocation:** Based on profiled demand, the controller invokes an online allocation algorithm to decide how many agent instances should be assigned to each type. This algorithm considers both current workload and the overhead associated with reconfiguration, aiming to optimize serving throughput. 3) **Agent Reconfiguration:** The controller dispatches updated allocation decisions to agent instances. Each instance then reconfigures itself to the new agent type by initializing the system prompt and migrating runtime state. To minimize latency, the reconfiguration process leverages two system-level optimizations: *consistency-aware prompt KV reuse*, which avoids redundant KV computation for consistent prompt segments; and *hybrid state KV migration*, which overlaps KV transmission and token-based recomputation to accelerate runtime state migration. Together, these components enable MAIA to serve multiple agent types adaptively, achieving high elasticity and efficient resource utilization.

B Context Sensitivity Quantification

To quantify this context sensitivity, we define a *sensitivity score* for each token, computed as the cosine distance between KV results with and without preceding context. As shown in Figure 4, the prompt KV reuse process involves the following steps:

1. Identify the shared prompt segments between the source and target agent types.
2. Compute the sensitivity scores of each token in the shared segment with respect to the source prompt. Mark tokens exceeding a threshold σ as legacy-influenced.

Algorithm 1: Agent instance allocation algorithm.

1 **Input:** $\epsilon > 0$ and $\eta = \ln(1 + N/\epsilon)$, where $N = |\mathcal{N}|$;
2 Initialize $x_i^t = 0, \forall i \in \mathcal{N}$;
3 Estimate reconfiguration cost $\beta_i, \forall i \in \mathcal{N}$;
4 **for** $t \in \mathcal{T}$ **do**
5 Solve the following optimization problem:

$$\min \sum_i f_i^t(x_i^t) + \frac{1}{\eta}\beta_i \left[\left(x_i^t + \frac{\epsilon}{N} \right) \cdot \ln \left(\frac{x_i^t + \epsilon/N}{\tilde{x}_i^{t-1} + \epsilon/N} \right) - x_i^t \right] \tag{6}$$

subject to: (2)

6 **Output:** Agent instance allocation decision $\tilde{x}_i^t$;

3. Repeat the sensitivity score computation using the target prompt as the preceding context. Mark tokens exceeding σ as context-missing.
4. Recompute the KV cache for all marked tokens, along with the non-overlapping segments of the target prompt. Merge the reused KV and recomputed KV to construct the full prompt KV cache.

Since system prompts are static and rarely change, MAIA performs this token-level sensitivity analysis offline. A lookup table is built to store the sensitivity scores for each token in every prompt segment. During reconfiguration, MAIA simply queries this table to identify which tokens require recomputation, avoiding runtime attention analysis. This design ensures that prompt KV reuse remains both fast and semantically reliable during agent reconfiguration.

C KV MIGRATION COST MODEL AND OPTIMAL PARTITIONING

MAIA formulates cost models for both operations. The computation cost $C_c(l)$ models the time required to generate the KV cache for the first l tokens of the state:

$$C_c(l) = \frac{r(al^2d + bld^2)}{P}, \tag{4}$$

where r is the number of transformer layers, d is the model's hidden dimension, and a, b are constants capturing the per-layer computational complexity. The term P indicates the GPU computing speed. The communication cost C_t models the time required to transmit the KV cache:

$$C_t(l) = \frac{2rlds}{B}, \tag{5}$$

where s is the per-element size in bytes, and B is the available bandwidth.

MAIA determines the optimal partition point l^* by solving the equation $C_c(l^*) = C_t(L - l^*)$, ensuring that the KV recomputation and KV transmission are well overlapped, where L is the total number of state tokens. This adaptive strategy enables MAIA to minimize the effective reconfiguration time across heterogeneous network and hardware conditions.

D ONLINE AGENT INSTANCE ALLOCATION ALGORITHM DESIGN

As shown in Algorithm 1, we first initialize instance allocations of all agent types to zero and set two system parameters: the learning rate η and a small smoothing constant ϵ. We then estimate the per-instance reconfiguration cost β_i for each agent type (Line 3). At each time slot t, we solve the revised optimization problem (6), where $f(\mathbf{x}^t)$ denotes the total serving throughput loss at time t, and the second term is a relative entropy regularization that penalizes large deviations in the

number of allocated instances compared to the previous time slot. The problem is convex and can be solved efficiently using standard methods Once solved, the output $\tilde{x}_i^t$ specifies the number of elastic instances assigned to each agent type i at time t. The system then executes the corresponding reconfiguration actions and proceeds to the next time slot.

E Algorithm Analysis

Definition 1. *The online algorithm with solution $\mathbf{X}_{ALG}$ is ρ-competitive if $\rho \geq \mathbf{X}_{ALG}/\hat{\mathbf{X}}_P$, where $\hat{\mathbf{X}}_P$ is the optimal solution given all information across time slots.*

Theorem 1. *The proposed online agent assignment algorithm is $(1+\frac{\beta^{max}MT-\Delta}{f^{min}NT+\Delta})$-competitive, where $\beta^{max} = \max_i\{\beta_i\}$, $f^{t,min} = \min_i\{f_i^t(\bar{S}_i^t)\}$, and $\Delta = \sum_{i,t}\frac{\beta_i\tilde{x}_i^t}{\eta}\ln(\frac{\tilde{x}_i^t+\epsilon/N}{\tilde{x}_i^{t-1}+\epsilon/N})$.*

Proof. We first simplify the above formulation as follows.

$$(P): \quad \min \sum_{i\in\mathcal{N}}\sum_{t\in\mathcal{T}} f_i^t(x_i^t) + \beta_i y_i^t, \quad \text{s.t.} \tag{7}$$

$$y_i^t \geq x_i^t - x_i^{t-1}, \forall i \in \mathcal{N}, t \in \mathcal{T}; \tag{8}$$

$$y_i^t \geq 0, \forall i \in \mathcal{N}, t \in \mathcal{T}, \tag{9}$$

$$\text{and (2)}$$

The Lagrangian formulation of the original problem (7) can be written as follows.

$$L(\lambda,\theta,\mu,x,y) = \max_{x,y}\min_{\lambda,\theta,\mu} P \tag{10}$$

$$= \sum_{i,t}(f_i^t(x_i^t) + \beta_i y_i^t) - \sum_{i,t}\lambda_i^t(y_i^t - x_i^t + x_i^{t-1})$$

$$+ \sum_t \theta_t(\sum_i x_i^t - M) - \sum_{i,t}\mu_i^t y_i^t \tag{11}$$

$$= \sum_{i,t} f_i^t(x_i^t) - \sum_{i,t}(\lambda_i^{t+1} - \lambda_i^t - \theta^t)x_i^t$$

$$+ \sum_{i,t}(\beta_i - \lambda_i^t - \mu_i^t)y_i^t - \sum_t \theta^t M \tag{12}$$

where λ_i^t, μ_i^t, θ^t, and φ_j^t are positive Lagrangian multiplier. Then, we can formulate the dual problem of the original problem as follows.

$$D(\lambda,\theta,\mu) = \min_{x,y} L(\lambda,\theta,\mu,x,y) \tag{13}$$

$$= \min_x \sum_{i,t} f_i^t(x_i^t) - (\lambda_i^{t+1} - \lambda_i^t - \theta^t)x_i^t$$

$$+ \min_y \sum_{i,t}(\beta_i - \lambda_i^t - \mu_i^t)y_i^t - \sum_t \theta^t M \tag{14}$$

Due to the weak duality, we have $\hat{\mathbf{X}}_D \leq \hat{\mathbf{X}}_P$, where $\hat{\mathbf{X}}_D$ and $\hat{\mathbf{X}}_P$ are optimal solutions of the dual problem $D(\lambda,\theta,\mu)$ and the original problem P. Then, we can get $\mathbf{X}_{ALG}/\hat{\mathbf{X}}_P \leq \mathbf{X}_{ALG}/\hat{\mathbf{X}}_D$. According to the KKT condition of optimization problem (6), we have $\sum_t \tilde{\theta}^t(\sum_i \tilde{x}_i^t - M) = 0$, where $\tilde{x}_i^t$ is the solution from the online algorithm at time slot t. In addition, $\tilde{\theta}^t$ is the solution of the Lagrangian form of the optimization problem (7). Then, by setting $\lambda_i^t = \frac{\beta_i}{\eta}\ln(\frac{1+\epsilon/N}{\tilde{x}_i^{t-1}+\epsilon/N})$, $\mu_i^t = 0$,

Setting	Optimal l^*	Optimal Lat. (s)	MAIA l^*	MAIA Lat. (s)	Error (%)
3090/300Mbps	703	3.18	678	3.27	+2.83%
3090/600Mbps	547	2.31	526	2.42	+4.76%
L20/300Mbps	832	2.56	815	2.63	+2.73%
L20/600Mbps	651	2.08	636	2.17	+4.32%

Table 1: The accuracy of the heuristic partitioning method.

and $\theta^t = \tilde{\theta}^t$, we have:

$$D(\lambda,\theta,\mu) = \min_x \left[\sum_{i,t} f_i^t(x_i^t) + (\frac{\beta_i}{\eta}\ln(\frac{\tilde{x}_i^t+\epsilon/N}{\tilde{x}_i^{t-1}+\epsilon/N}) - \tilde{\theta}^t)x_i^t\right] + \min_y \sum_{i,t}(\beta_i - \lambda_i^t - \mu_i^t)y_i^t - \sum_t \tilde{\theta}^t M \tag{15}$$

$$= \sum_{i,t} f_i^t(\tilde{x}_i^t) + \frac{\beta_i}{\eta}\ln(\frac{\tilde{x}_i^t+\epsilon/N}{\tilde{x}_i^{t-1}+\epsilon/N})\tilde{x}_i^t. \tag{16}$$

Then, from the weak duality, we have:

$$\frac{\mathbf{X}_{ALG}}{\hat{\mathbf{X}}_P} \le \frac{\sum_{i,t} f_i^t(\tilde{x}_i^t) + \beta_i \tilde{y}_i^t}{\sum_{i,t} f_i^t(\tilde{x}_i^t) + \frac{\beta_i}{\eta}\ln(\frac{\tilde{x}_i^t+\epsilon/N}{\tilde{x}_i^{t-1}+\epsilon/N})\tilde{x}_i^t} \tag{17}$$

$$\le 1 + \frac{\sum_{i,t}\beta_i M - \frac{\beta_i}{\eta}\ln(\frac{\tilde{x}_i^t+\epsilon/N}{\tilde{x}_i^{t-1}+\epsilon/N})\tilde{x}_i^t}{\sum_{i,t} f_i^t(\tilde{x}_i^t) + \frac{\beta_i}{\eta}\ln(\frac{\tilde{x}_i^t+\epsilon/N}{\tilde{x}_i^{t-1}+\epsilon/N})\tilde{x}_i^t} \tag{18}$$

$$\le 1 + \frac{\beta^{\max} MT - \Delta}{f^{\min} NT + \Delta}, \tag{19}$$

where $\Delta = \sum_{i,t}\frac{\beta_i \tilde{x}_i^t}{\eta}\ln(\frac{\tilde{x}_i^t+\epsilon/N}{\tilde{x}_i^{t-1}+\epsilon/N})$. □

F SUPPLEMENTARY EXPERIMENTS

Impact of resource configurations. We then study the effect of system resource configurations on migration performance. We consider different levels of communication and computation capabilities, with combinations labeled as WW (weak communication + weak computation) through SS (strong communication + strong computation). We use 300Mbps and 600Mbps to represent weak and strong communication bandwidths, and NVIDIA 3090 and L20 GPUs to represent computation resources. As shown in Figure 9(c), the KV transmission method benefits most from higher bandwidth, while the KV recomputation method benefits from more powerful GPUs. In contrast, our hybrid method is able to leverage both resources simultaneously. On average, it reduces migration latency by up to 45%.

In addition, the effectiveness of hybrid state KV migration depends on accurately selecting the partition point l^*, which determines how much runtime state should be recomputed from tokens and how much should be directly transferred as KV cache. In principle, the optimal l^* can be obtained by exhaustively testing all possible partition points and selecting the one with the minimum end-to-end migration latency. However, such an exhaustive search is impractical for online serving, as it would introduce additional runtime overhead and require repeated profiling under changing hardware and network conditions. To enable efficient online decision-making, MAIA adopts a lightweight heuristic based on the analytical cost model in §3.2. The heuristic selects l^* by balancing the estimated recomputation cost and KV transmission cost, thereby avoiding expensive runtime search. Our empirical results show that it is sufficiently accurate in practice. To validate this, we compare MAIA's selected partition point with the true optimal partition obtained by exhaustive search. We conduct experiments using 1K-token runtime states under four representative GPU/network configurations, including two GPU types and two bandwidth settings. As shown in Table 1, MAIA's partition

heuristic consistently produces latency close to the true optimum. Across all settings, the latency error remains within 4.76%, with an average error of only 3.66%. This demonstrates that the proposed heuristic can effectively approximate the optimal partition while avoiding the overhead of online exhaustive search, making it suitable for practical agent serving systems.

Simulator fidelity.

We use a trace-driven simulator to evaluate MAIA at scale, which is calibrated using profiling results collected from a real testbed. To validate both the fidelity of the simulator and the effectiveness of MAIA in a real deployment, we further conduct an end-to-end evaluation on a testbed with $8\times$ NVIDIA L20 GPUs. The system serves LLaMA3-8B agents using vLLM, where each agent instance occupies one GPU and the batch size is set to 1. We sample 10-hour request traces from the WildChat dataset and construct workloads from two agent domains. The results are reported in Table 2. The testbed results show that MAIA provides substantial performance improvement over the default static serving strategy. Without MAIA, agent instances are statically bound to agent types and cannot be adjusted according to workload shifts. As a result, some agent types become overloaded while others remain underutilized, leading to both low throughput and high tail latency. In contrast, MAIA dynamically reconfigures agent instances to better match time-varying demand. On the testbed, MAIA improves throughput from 60.84 to 121.58 reqs/hour, achieving about $2\times$ throughput. Meanwhile, it reduces P50, P90, and P99 latency by 60.8%, 67.0%, and 64.2%, respectively. The simulator also closely matches the real testbed results. Compared with the testbed result under MAIA, the simulator reports only a small performance gap: the throughput differs by about 5.4%, and the P50, P90, and P99 latency differ by 7.8%, 7.6%, and 8.9%, respectively. This indicates that the trace-driven simulator can faithfully capture the major performance trends of MAIA, making it suitable for evaluating larger-scale settings that are difficult to deploy directly on the physical testbed.

Table 2: Testbed and simulator performance.

Setting	Throughput (req/h)	P50 (s)	P90 (s)	P99 (s)
Testbed (w/o MAIA)	60.84	124.11	159.47	182.55
Testbed (w/ MAIA)	121.58	48.64	52.64	65.34
Simulator	128.11	44.84	48.66	59.54

MAIA Overhead. The main additional overhead introduced by MAIA comes from the online instance allocation algorithm. We therefore measure its execution time on CPU under different system scales by varying both the number of agent types and the number of elastic instances. As shown in Table 3, the overhead increases with system scale, since larger configurations involve a larger allocation space. Nevertheless, the runtime remains at the millisecond level across all evaluated settings. Even in the largest case with 32 agent types and 90 instances, the allocation algorithm takes only 136.47 ms. This overhead is negligible in practice for two reasons. First, the online allocation algorithm is invoked periodically rather than on a per-request basis, so its cost is amortized over many inference requests within each scheduling interval. Second, the measured scheduling time is much smaller than the latency of agent reconfiguration and LLM inference, which typically occurs at the second scale. Therefore, the online allocation algorithm does not become a system bottleneck, while enabling MAIA to make workload-aware and cost-aware reconfiguration decisions efficiently.

Table 3: Overhead of MAIA (ms).

#Agent Types	#Instances				
	10	30	50	70	90
2	28.51	32.84	37.29	48.16	59.63
4	33.62	41.33	50.24	61.87	73.15
8	40.17	51.12	62.98	73.92	86.47
16	49.28	62.77	78.69	95.72	118.36
32	63.47	75.15	88.94	108.47	136.47

G SUPPLEMENTARY RELATED WORK

LLM-based Agents. LLM-based agents leverage large language models to support intelligent decision-making and autonomous task execution. By augmenting LLMs with domain-specific adapters, structured system prompts, external tools, and runtime states, these agents can perform complex interactive tasks beyond single-turn text generation. Prior studies have explored various capabilities of LLM-based agents, including multi-turn reasoning (Yao et al., 2023b), tool us-

age (Nakano et al., 2021), long-horizon planning (Zhang et al., 2025), and self-reflection or policy refinement (Guo et al., 2025). Representative systems such as WebGPT (Nakano et al., 2021), AutoGPT (Yang et al., 2023) and AgentGPT (Liu et al., 2024b). Beyond single-agent systems, recent works have investigated multi-agent collaboration, where multiple LLM-based agents interact to solve complex tasks through role specialization and communication (Wang et al., 2025a). For example, CAMEL (Li et al., 2023) studies role-playing agents for cooperative problem solving, AutoGen (Wu et al., 2024) enables flexible multi-agent conversations, and MetaGPT (Hong et al., 2024) organizes agents with software-engineering-style roles. Other works focus on improving collaboration topologies, communication protocols, and debate or aggregation mechanisms. These studies primarily aim to improve agent capability, reasoning quality, and collaboration effectiveness. In contrast, MAIA focuses on the system-level serving problem: how to efficiently deploy and dynamically reconfigure heterogeneous agent instances under fluctuating workloads.

Agent Serving Systems. Agent serving systems follow the Agent-as-a-Service paradigm to concurrently serve diverse user requests by deploying multiple types of agents (Lin et al., 2024; Wu et al., 2026; Ray et al., 2025). Existing systems have explored different aspects of agent execution and scheduling. For example, AIOS (Mei et al., 2024) proposes an operating-system-like framework to coordinate LLM agents and manage inference requests. AutoGen (Wu et al., 2024) and CAMEL (Li et al., 2023) emphasize task orchestration and inter-agent communication. Agentix (Luo et al., 2026) studies efficient serving of LLM agents as general programs. Parrot (Lin et al., 2024) improves resource utilization by scheduling LLM-based application tasks and exploiting semantic information in shared prefixes. Amayuelas et al. (2025) further explore self-resource allocation, where agents autonomously reason about their own resource usage. Despite these advances, many existing agent serving systems rely on static agent-to-instance bindings. Each agent instance is initialized with a fixed agent type and remains unchanged throughout its lifetime. Such designs are simple to implement but lack elasticity under dynamic workloads, where demand for different agent types can fluctuate significantly over time. Static bindings may therefore cause under-provisioning for overloaded agent types and resource waste for underutilized ones. Recent works have begun to address workload dynamicity in agent serving. KVFlow (Pan et al., 2026) proposes an adaptive prompt caching mechanism to accelerate system prompt initialization across agent workflows. Zhang et al. (2024) study topology-aware preemptive scheduling for co-located LLM workloads. These approaches provide useful mechanisms for improving responsiveness, but they either overlook the full overhead of agent reconfiguration, such as prompt KV initialization and runtime state migration, or lack a theoretically grounded online allocation algorithm for uncertain workloads.

MAIA differs from prior work in two aspects. First, it explicitly optimizes agent reconfiguration overhead by combining consistency-aware prompt KV reuse with hybrid state KV migration. Second, it formulates dynamic agent instance allocation as an online optimization problem and provides a performance guarantee, enabling cost-aware reconfiguration decisions under unpredictable workload shifts. As a result, MAIA provides a unified system design that jointly addresses fast agent reconfiguration and online resource allocation.