AIRMAS: INTER-AGENT INTERMEDIATE ACTIVATION REUSE FOR LORA-BASED MULTI-AGENT SYSTEM

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM-based multi-agent systems assign a different role to each agent. Fine-tuning a separate model for every role is expensive in both compute and storage. To reduce the burden, recent systems share one frozen base model and give each agent an additional role-specific LoRA adapter. When the agents run in sequence, each agent receives the context produced so far and appends its own output. Because every adapter transforms the same tokens differently, each agent prefills the shared context again and keeps its own KV cache. Prefill cost and memory therefore increase with the context length. To address this issue, recent work splits the value cache into two parts. All agents share the base cache computed with the frozen backbone weights, and each agent keeps a small low-rank part from its own LoRA adapter. Reusing the low-rank part across agents requires every adapter to share a single down-projection. Adapters with independently trained down-projections cannot reuse it. We propose AIRMAS, an inference framework that addresses this limitation. Instead of sharing a single down-projection matrix, AIRMAS maps the stored low-rank cache of an earlier agent to the current agent with an $r \times r$ matrix precomputed from the two down-projections. On agentic question-answering benchmarks, AIRMAS improves accuracy by up to 3.33% over BaseLRShared, the low-rank cache sharing scheme of LRAgent. It requires no change to the adapters, and its time-to-first-token stays within 2.4% of that scheme.

1 INTRODUCTION

Large language models (LLMs) support long-context understanding (Dubey et al., 2024), multi-step reasoning (Wei et al., 2022), and external tool use (Yao et al., 2023; Qin et al., 2024). However, a single model struggles with tasks that require both multi-step reasoning and domain-specific expertise (Shen et al., 2024). To address this issue, multi-agent systems assign specialized roles to different agents (Talebirad & Nadiri, 2023; Wu et al., 2024; Hong et al., 2024). These roles include planning, tool use, and reflection. The agents divide a complex task into multiple steps and solve it collaboratively. Such systems have shown higher task accuracy than single-model baselines (Shen et al., 2024; Qiao et al., 2024). In a sequential multi-agent system, each agent receives the accumulated context and performs its distinct role. It then passes its output to the next agent. Agents may also call external tools, such as web search or code execution (Yao et al., 2023; Qin et al., 2024). Their outputs and tool responses form an accumulating trajectory (Qiao et al., 2024; Jeon et al., 2026). Role-based fine-tuning trains agents on role-specific trajectories to improve their performance on specific tasks (Qiao et al., 2024; Yu et al., 2024; Park & Kwon, 2026). However, fine-tuning and storing a separate full model for each role requires substantial computation and memory.

Parameter-efficient fine-tuning (PEFT) reduces the cost of adaptation. Low-rank adaptation (LoRA) (Hu et al., 2022) freezes the pretrained weight W_0 and learns a low-rank update $\Delta W = AB$. Here, $A \in \mathbb{R}^{d_{\text{in}} \times r}$ is the down-projection matrix, and $B \in \mathbb{R}^{r \times d_{\text{out}}}$ is the up-projection matrix. The rank r is much smaller than d_{in} and d_{out}. Only A and B are trained, which reduces the number of trainable parameters and the storage required for each adapted model. Multi-LoRA systems use this property by sharing the pretrained backbone across multiple adapters (Wang et al., 2023; Sheng et al., 2024; Xia et al., 2024). This design supports role specialization with a small set of additional

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.

parameters per agent (Qiao et al., 2024; Jeon et al., 2026). Sharing model weights, however, does not eliminate repeated prefill or duplicated KV caches. As shown in Figure 1(a), a receiving agent computes the prefix tokens that are not available in its own cache. As a result, longer trajectories increase prefill latency, and separate caches for the agents increase the memory footprint (Kwon et al., 2023; Woo et al., 2026). Existing methods reduce repeated prefill through KV cache reuse (Kwon et al., 2023; Zheng et al., 2024). However, LoRA adapters can produce different KV caches for the same input contexts. To reduce this redundancy, LRAgent (Jeon et al., 2026) separates the value cache into a base cache and a low-rank (LR) cache and shares them across agents. Its LR-cache sharing scheme requires all agents to use the same down-projection matrix A. With a shared A, the agents differ only in their up-projections B. A single shared down-projection can limit specialization (Zuo et al., 2026). Adapters with independently trained down-projections also cannot use this scheme. We present AIRMAS, an efficient LoRA-based multi-agent inference framework that reuses LR caches across agents with independently trained adapters. Rather than directly sharing down-projection weights, we approximate the target LoRA intermediate activations by transforming the source LR cache. The transformation is an $r \times r$ matrix $T_{i\to j} = A_i^\dagger A_j$. Here, $A_i^\dagger$ denotes the pseudo-inverse of the source down-projection matrix. AIRMAS thus reuses intermediate activations across agents (Figure 1(b)). BAIRMAS retains each agent's down-projection and is more accurate than the LR-cache sharing scheme of LRAgent. AIRMAS requires no changes to the model architecture or training procedure. Our contributions are threefold:

- **Inter-agent intermediate activation reuse.** We introduce a cache-sharing framework that reuses LoRA down-projection activations across agents with independently trained adapters. It concatenates the transformed suffix LR caches of the earlier agents in token order, so the current agent does not recompute the prefix hidden states.
- **Accuracy-recovering lightweight pseudo-inverse transformation.** We reduce the accuracy drop from reusing another agent's intermediate activation with a lightweight transformation. We derive an $r \times r$ transformation $T_{i\to j} = A_i^\dagger A_j$ between the LR caches of two agents. It is precomputed from the adapter weights for each layer and requires $\mathcal{O}(r^2)$ operations per transformed token. With this transformation, AIRMAS is up to 3.33% more accurate than `BaseLRShared` (Jeon et al., 2026).
- **Higher accuracy with comparable inference time using fused kernel.** A separate transformation kernel writes the transformed LR cache to global memory before Flash-LoRA-Attention (Jeon et al., 2026) reads it. We fuse the transformation into Flash-LoRA-Attention, so the transformed LR cache is never written to global memory. The TTFT of AIRMAS stays within 2.4% of `BaseLRShared`. Because only the suffix tokens are prefilled, AIRMAS reaches up to $1.78\times$ the throughput of `Non-Shared`.

2 METHODOLOGY

2.1 OVERVIEW

Prefix reuse in a sequential multi-LoRA agent system. Figure 1 considers n agents invoked once in sequence. They share a frozen backbone and hold role-specific LoRA adapters $\{(A_i, B_i)\}_{i=0}^{n-1}$, where A_i is the down-projection of agent i. Each adapter is trained for its own assigned role. Each agent receives the accumulated trajectory from the previous agent, performs its designated role and appends its output. As a result, the context of agent $n-1$ consists of $L_{p_{n-1}}$ prefix tokens produced by the earlier agents and its own $L_{s_{n-1}}$ suffix tokens. The suffix of agent i consists of the L_{s_i} input tokens that agent i prefills and the tokens that it generates. In this example, each agent is invoked for the first time and has no cache from an earlier invocation of its own. Without cross-agent reuse, each agent therefore builds its cache over the whole accumulated context. In the general case, an agent can be invoked repeatedly and reuse the common prefix of its own cache (Section 3.1).

Cache decomposition and redundant computation. The right part of Figure 1(a) shows why the stored caches cannot be reused directly. At a given layer, let W_0 denote the frozen value-projection matrix. Let $X_{n-1} \in \mathbb{R}^{(L_{p_{n-1}}+L_{s_{n-1}})\times d_{in}}$ denote the hidden state of agent $n-1$ and let r be the LoRA rank. The value cache is

$$F_{n-1} = X_{n-1}W_0 + (X_{n-1}A_{n-1})B_{n-1} = P_{n-1} + Z_{n-1}B_{n-1}, \tag{1}$$

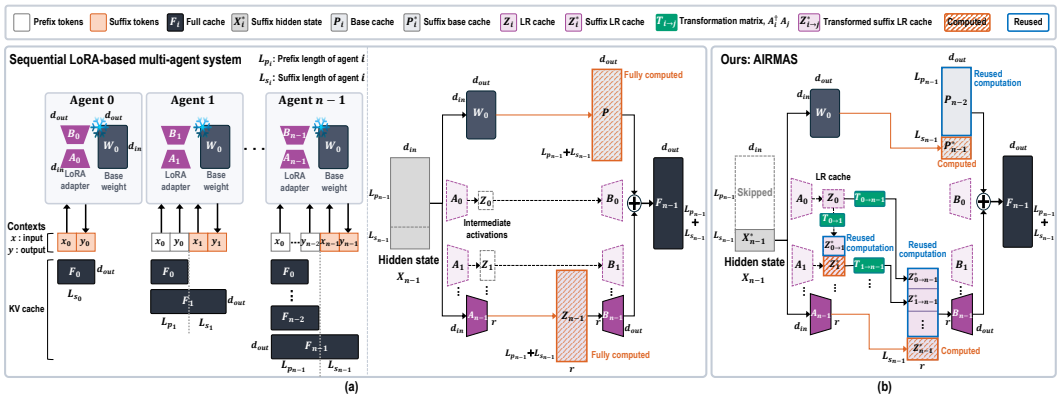

Figure 1: Computational flow of a sequential LoRA-based multi-agent system. All agents share the frozen backbone, whose value projection at a layer is W_0, but each holds an independently trained LoRA adapter (A_i, B_i). The previous agent forwards its accumulated context of L_{p_i} tokens as the prefix of the next agent i. Agent i appends its own L_{s_i} suffix tokens and runs inference. (a) The conventional method recomputes all the intermediate activations (Z_i) and builds both the base cache P_i and the LR cache Z_i for all tokens $L_{p_i} + L_{s_i}$. (b) AIRMAS reuses the base cache and transforms the stored suffix LR caches with the precomputed $T_{i\to j}$. It computes intermediate activations only for the L_{s_i} suffix tokens. Dashed adapters are not used by agent $n-1$.

where $P_{n-1} = X_{n-1}W_0$ is the base cache and $Z_{n-1} = X_{n-1}A_{n-1} \in \mathbb{R}^{(L_{p_{n-1}}+L_{s_{n-1}})\times r}$ is the low-rank (LR) cache (Jeon et al., 2026). With LoRA on the query and value projections, the key cache has no adapter. We reuse it across agents in the same way as the base cache. The base cache applies the shared value-projection matrix W_0 to the hidden states. In contrast, the LR cache is produced by each agent's down-projection matrix, so the intermediate activations from the earlier agents do not match the ones agent $n-1$ requires. In this example, the conventional method recomputes the hidden state X_{n-1} and both P_{n-1} and Z_{n-1} over all $L_{p_{n-1}} + L_{s_{n-1}}$ tokens. This cost increases with the prefix length.

Inter-agent intermediate activation reuse. Figure 1(b) shows how AIRMAS removes this redundant computation. Since the backbone is shared, agent $n-1$ reuses the base cache P_{n-2} that the previous agent has already stored. For the adapter path, agent $n-1$ receives the suffix LR cache Z_i^* of each earlier agent i, which is the intermediate activation that the adapter of agent i produced. AIRMAS transforms each Z_i^* with the precomputed matrix $T_{i\to n-1}$, which depends only on the down-projections of agents i and $n-1$. The transformed segments form the prefix of $\widehat{Z}_{n-1}$. Agent $n-1$ computes hidden states only for its own $L_{s_{n-1}}$ suffix tokens.

2.2 LR CACHE CONSTRUCTION ACROSS AGENTS

LR cache construction in the previous schemes. `BaseShared` (Figure 2(a)) keeps a separate A_i for every agent and shares only the base cache. Agent $n-1$ must therefore compute $Z_{n-1} = X_{n-1}A_{n-1}$ over all $L_{p_{n-1}} + L_{s_{n-1}}$ tokens. This requires the prefix hidden states, and the forward pass over the prefix cannot be skipped. `BaseLRShared` (Figure 2(b)) removes this forward pass by forcing all agents to use a single shared down-projection A. With a shared A, the LR cache of the previous agent differs from the prefix LR cache of the current agent only through the hidden states. Agent $n-1$ constructs $Z_{n-1} = [\, Z_{n-2}; X_{n-1}^* A \,]$ and computes only the $L_{s_{n-1}}$ suffix tokens. This saving requires a shared down-projection, so adapters with independently trained down-projections cannot use it.

Transformed suffix LR cache. AIRMAS (Figure 2(c)) removes the prefix forward pass while keeping each A_i. The stored suffix LR caches of the earlier agents, $Z_0^*, \ldots, Z_{n-2}^*$, are transformed for agent $n-1$ as

$$Z_{i\to n-1}^* = Z_i^* T_{i\to n-1} \in \mathbb{R}^{L_{s_i}\times r}, \qquad i = 0, \ldots, n-2, \tag{2}$$

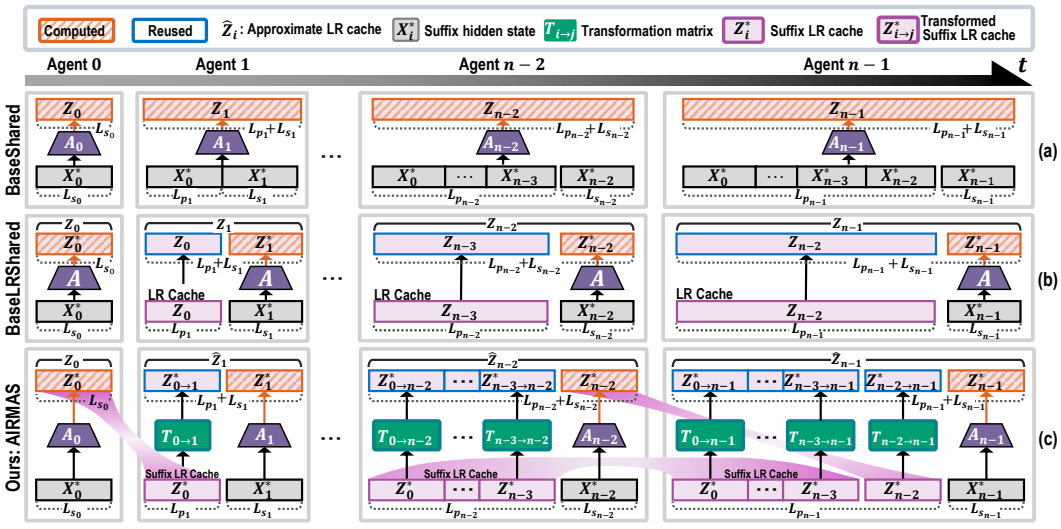

Figure 2: LR cache construction along the agent sequence, where each column is an agent in its invocation order. (a) BaseShared recomputes the whole LR cache with its own A_i over all $L_{p_i} + L_{s_i}$ tokens for every agent i. (b) BaseLRShared reuses the LR cache of the previous agent and computes only the L_{s_i} suffix tokens, but requires a single A shared across all agents. (c) AIRMAS also reuses the LR cache while keeping each A_i. To recover accuracy, the stored suffix LR cache Z_i^* of each earlier agent i is transformed into $Z_{i\to n-1}^*$ by the transformation matrix $T_{i\to n-1}$ to form $\widehat{Z}_{n-1}$ of the current agent $n-1$.

where the transformation matrix $T_{i\to n-1} \in \mathbb{R}^{r\times r}$ is precomputed from the adapter weights (Section 2.3). For its own $L_{s_{n-1}}$ suffix tokens, agent $n-1$ computes $Z_{n-1}^* = X_{n-1}^* A_{n-1}$ with its own adapter. Concatenating the transformed suffix LR caches and the newly computed one in token order gives the approximate LR cache

$$\widehat{Z}_{n-1} = [\, Z_{0\to n-1}^*; Z_{1\to n-1}^*; \ldots; Z_{n-2\to n-1}^*; Z_{n-1}^* \,] \in \mathbb{R}^{(L_{p_{n-1}}+L_{s_{n-1}})\times r}, \tag{3}$$

which replaces Z_{n-1} in Eq. 1. When an adapter is used again, the segments that it produced earlier are reused without transformation ($T = I$). Every agent thus retains its own A_i. Section 3.2 compares the accuracy of AIRMAS with `BaseLRShared`.

2.3 Lightweight pseudo-inverse transformation

Transformation matrix for prefix reuse. The left side of Figure 3 compares our lightweight pseudo-inverse transform with the conventional method. The conventional method applies A_j to the prefix hidden states, so it must recompute them. The exact prefix segment of agent j is $X_j A_j$, where X_j is the hidden state that agent j would compute for these tokens. AIRMAS makes two approximations. It first replaces X_j by the source hidden state X_i^*, as the reuse of the base cache does (Jeon et al., 2026). Computing $X_i^* A_j$ still requires X_i^*, which is not stored. AIRMAS instead reuses the stored suffix LR cache Z_i^* and approximates the segment by $Z_i^* T_{i\to j}$. We seek a transformation matrix such that

$$Z_i^* T_{i\to j} \approx X_i^* A_j, \tag{4}$$

so that X_i^* is not needed when agent j runs.

❶ Offline $T_{i\to j}$ precomputation. Substituting $Z_i^* = X_i^* A_i$ into Eq. 4 gives $X_i^* A_i T_{i\to j} \approx X_i^* A_j$. We choose the transformation from the adapter weights by minimizing $\|A_i T - A_j\|_F^2$. The minimum-norm least-squares solution is obtained with the pseudo-inverse $A_i^\dagger \in \mathbb{R}^{r\times d_{in}}$ of the source down-projection as

$$T_{i\to j} = A_i^\dagger A_j \in \mathbb{R}^{r\times r}. \tag{5}$$

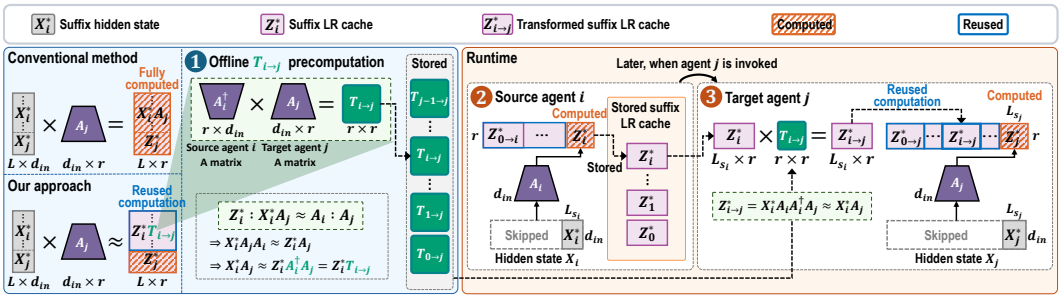

Figure 3: Lightweight pseudo-inverse transform. ❶ Offline, we precompute $T_{i\to j} = A_i^\dagger A_j \in \mathbb{R}^{r\times r}$ for every agent pair and layer and store it. ❷ At runtime, source agent i computes the hidden state X_i^* only for its own L_{s_i} suffix tokens. It stores the suffix LR cache $Z_i^* = X_i^* A_i$. ❸ When target agent j is invoked later, it multiplies the stored Z_i^* by $T_{i\to j}$. The result $Z_{i\to j}^* = X_i^* A_i A_i^\dagger A_j$ approximates $X_i^* A_j$ without X_i^*. The transformed segments are concatenated with the Z_j^* that agent j computes for its L_{s_j} suffix tokens.

Table 1: Comparison of memory footprint and computational complexity with previous cache-sharing schemes. N is the number of adapters, $L = L_{p_i} + L_{s_i}$ is the accumulated trajectory length of agent i and r is the LoRA rank where $r \ll d_{\text{in}}, d_{\text{out}}$. d_{in} and d_{out} are the input and output dimensions of the value projection W_0. $C = d_{in}d_{out} + Ld_{out}$ is the per-token cost of the value projection and the attention over L. The Lr^2 term indicates the transformation cost for the stored suffix LR caches. The accuracy column summarizes Table 2. For LLaMA-3.1-8B, the value projection has $d_{\text{in}} = 4096$ and $d_{\text{out}} = 1024$. The shared backbone has $M \approx 8 \times 10^9$ parameters. In our experiments, we use $N = 3$ role-specific adapters with LoRA rank $r = 8$. Except for the first and the retrieval steps, the suffix lengths L_{s_i} are on the order of 10^1. On the benchmarks, L is on the order of 10^3 (Appendix D.4). Prefill counts the L_{s_i} input tokens of the suffix, and we write L_{s_i} for brevity. C is on the order of 10^6 to 10^8. Appendix B expands these terms.

	Memory	Computation		
Scheme	Caches and transformation matrices	Prefill (per agent)	Decode (per token)	Accuracy ↑
`Non-Shared`	$\mathcal{O}(NLd_{\text{out}})$	$\mathcal{O}(LC)$	$\mathcal{O}(C)$	★ Best
`BaseShared`	$\mathcal{O}(Ld_{\text{out}} + NLr)$	$\mathcal{O}(LC)$	$\mathcal{O}(C)$	✓ Good
`BaseLRShared`	$\mathcal{O}(Ld_{\text{out}} + Lr)$	$\mathcal{O}(L_{s_i}C)$	$\mathcal{O}(C)$	△ Moderate
AIRMAS	$\mathcal{O}(Ld_{\text{out}} + Lr + N^2r^2)$	$\mathcal{O}(L_{s_i}C + L_{s_i}Lr^2)$	$\mathcal{O}(C + Lr^2)$	✓ Good

The product $A_iA_i^\dagger$ is the orthogonal projector onto the column space of A_i. $Z_i^*T_{i\to j}$ equals $X_i^*A_j$ when $X_i^*(I - A_iA_i^\dagger)A_j = 0$. Eq. 5 involves only the adapter weights. We therefore precompute $T_{i\to j}$ once for every agent pair and layer. Each $T_{i\to j}$ is an $r \times r$ matrix.

❷❸ Runtime suffix LR cache transformation. ❷ When source agent i is invoked, it forwards only its own L_{s_i} suffix tokens. It stores the resulting suffix LR cache $Z_i^* = X_i^*A_i$. ❸ Later, when target agent j is invoked, the stored Z_i^* is multiplied by the precomputed $T_{i\to j}$ to give

$$Z_{i\to j}^* = Z_i^*T_{i\to j} = X_i^*A_iA_i^\dagger A_j \approx X_i^*A_j \in \mathbb{R}^{L_{s_i}\times r}, \tag{6}$$

which approximates the LR cache of the corresponding prefix segment for agent j. The transformed segments are concatenated with the Z_j^* that agent j computes for its own L_{s_j} suffix tokens. Each segment is transformed directly from its stored Z_i^*. For example, agent 2 obtains the segment of agent 0 as $Z_0^*T_{0\to 2}$ rather than $Z_{0\to 1}^*T_{1\to 2}$. Transforming one token requires $r \times r$ multiply-adds. AIRMAS does not store the transformed suffix LR cache $Z_{i\to n-1}^*$, so the transformation is repeated at every decoding step (Section 2.5). It adds at most Lr^2 multiply-adds per head and layer at every decoding step. For $r = 8$ this is negligible compared with the per-token cost C of the value projection and the attention over the L tokens (Table 1).

2.4 MEMORY AND COMPUTATION

Table 1 compares the memory and computation of `Non-Shared`, `BaseShared`, `BaseLRShared` and AIRMAS with the notation of its caption. The memory column includes the stored KV cache and the transformation matrices. The prefill column counts the computation of agent i before its first generated token, and the decode column counts the computation per generated token. Since $r \ll d_{\text{in}}, d_{\text{out}}$, the LoRA terms $d_{\text{in}}r$, rd_{out} and Lr are dominated by C and omitted. We keep the Lr^2 term because only AIRMAS requires it. Appendix B expands these terms.

All schemes share the pretrained weight W_0. Their weight parameters are $\mathcal{O}(M + Nrd_{\text{in}} + Nrd_{\text{out}})$ with independently trained adapters and $\mathcal{O}(M + rd_{\text{in}} + Nrd_{\text{out}})$ with a shared down-projection A across agents. `Non-Shared` maintains a full cache F_i for each agent i without sharing. Its cache memory is $\mathcal{O}(NLd_{\text{out}})$. Different adapters produce different KV caches for the same tokens. Hence the current agent cannot reuse the caches of the other agents. It prefills every token outside the longest common prefix of its own cache, which is $\mathcal{O}(LC)$ in the worst case. `BaseShared` shares only the base cache P_i and keeps a separate A_i for every agent. Agent i recomputes the whole LR cache $Z_i = X_iA_i$ over all L tokens. Its cache memory is $\mathcal{O}(Ld_{\text{out}} + NLr)$. The prefix LR cache requires the prefix hidden states. The forward pass over the prefix cannot be skipped and the prefill computation remains $\mathcal{O}(LC)$ (Jeon et al., 2026). `BaseLRShared` shares both the base cache P_i and the LR cache Z_i. Its cache memory is $\mathcal{O}(Ld_{\text{out}} + Lr)$. Only the L_{s_i} suffix tokens require hidden states and the prefill computation is reduced to $\mathcal{O}(L_{s_i}C)$. This requires a single down-projection matrix A for all agents. AIRMAS shares the base cache and stores the suffix LR cache Z_i^* for each agent. Its cache memory is $\mathcal{O}(Ld_{\text{out}} + Lr)$ as in `BaseLRShared`. The transformation matrices for all agent pairs add $\mathcal{O}(N^2r^2)$ memory. Hidden states are computed only for the L_{s_i} suffix tokens while each agent retains its own A_i. It requires no additional LR cache. The only additional computation is the transformation of the stored suffix LR caches of the earlier agents. It requires at most Lr^2 multiply-adds per head and layer at every decoding step. The kernel of Section 2.5 fuses this transformation into Flash-LoRA-Attention. The fusion removes the separate transformation kernel and its global memory traffic.

2.5 EFFICIENT ATTENTION COMPUTATION WITH THE LR CACHE

Flash-LoRA-Attention. $P_{n-1} \in \mathbb{R}^{L \times d_{\text{out}}}$ and $Z_{n-1} \in \mathbb{R}^{L \times r}$ denote the base cache and the LR cache in Eq. 1, and $B_{n-1} \in \mathbb{R}^{r \times d_{\text{out}}}$ is the up-projection. Under grouped-query attention (Ainslie et al., 2023), let H_q and H_{kv} denote the numbers of query and key/value heads, respectively, with $d_{\text{out}} = H_{kv}d_{\text{head}}$. For query head h, let $g(h)$ be its corresponding key/value head. The column blocks $P_{n-1}^{[g(h)]} \in \mathbb{R}^{L \times d_{\text{head}}}$ and $B_{n-1}^{[g(h)]} \in \mathbb{R}^{r \times d_{\text{head}}}$ belong to that key/value head. The LR cache has no head dimension, so all query heads read the same Z_{n-1}. Only B_{n-1} is split by key/value head. With attention weights $S_{n-1}^{[h]} \in \mathbb{R}^{L_{s_{n-1}} \times L}$, a straightforward implementation computes

$$O_{n-1}^{[h]} = S_{n-1}^{[h]} \left(P_{n-1}^{[g(h)]} + Z_{n-1}B_{n-1}^{[g(h)]} \right). \tag{7}$$

It applies the up-projection to all L tokens and materializes an $L \times d_{\text{head}}$ adapter output for each key/value head. Flash-LoRA-Attention reorders the matrix multiplications (Jeon et al., 2026):

$$O_{n-1}^{[h]} = S_{n-1}^{[h]}P_{n-1}^{[g(h)]} + \left(S_{n-1}^{[h]}Z_{n-1} \right) B_{n-1}^{[g(h)]}. \tag{8}$$

It first applies attention to the LR cache. The attention-weighted LR cache has shape $L_{s_{n-1}} \times r$, so the up-projection costs $\mathcal{O}(L_{s_{n-1}}rd_{\text{head}})$ per query head. In decoding, attention has one query row and this cost is $\mathcal{O}(rd_{\text{head}})$. The full-dimensional adapter output is never materialized. The base-cache attention remains unchanged. Table 1 treats H_q, H_{kv}, and their ratio as constants. We omit the head indices below.

Flash-LoRA-Attention with on-the-fly transformation. AIRMAS replaces Z_{n-1} by the approximate LR cache $\widehat{Z}_{n-1}$ in Eq. 3. A straightforward implementation transforms the stored suffix LR caches by $T_{i \to n-1}$ in a separate kernel. This implementation writes $\widehat{Z}_{n-1}$ to global memory, and Flash-LoRA-Attention reads it again. AIRMAS does not store $\widehat{Z}_{n-1}$, so this is repeated at every

decoding step. In our measurement the separate step adds about 15 ms to every decoding step (Appendix C.2). This cost does not increase with the trajectory length. It is therefore not explained by the size of $\widehat{Z}_{n-1}$. We remove it with a fused kernel. When a key/value block is loaded into shared memory, each row of the stored suffix LR caches is multiplied by the transformation matrix of the agent that produced the row. The transformed block directly enters the streaming accumulation of Flash-LoRA-Attention. $\widehat{Z}_{n-1}$ never enters global memory, and the kernel computes Eq. 8 with $\widehat{Z}_{n-1}$ without further approximation. Appendix C.1 gives the data layout and the pseudocode (Algorithm 1).

3 Evaluation

3.1 Experimental setup

Agent setup. We evaluate AIRMAS in the sequential multi-agent system of AutoAct (Qiao et al., 2024). Three agents share one frozen backbone and are invoked in sequence. The plan agent reasons over the accumulated trajectory and selects an external tool. The action agent produces the tool arguments and executes the call. After a Finish call, the reflect agent reviews the proposed answer and decides whether to terminate or resume planning. Each agent holds its own role-specific adapter so $N = 3$. Agent prompts and trajectory templates are provided in Appendix D.1.

Models and datasets. We evaluate on LLaMA-3.1-8B-Instruct (Dubey et al., 2024) and Ministral-8B-Instruct (Mistral AI, 2024). We fine-tune and evaluate on HotpotQA (Yang et al., 2018) and ScienceQA (Lu et al., 2022). HotpotQA evaluates multi-hop question answering. ScienceQA evaluates multiple-choice science question answering, including questions with visual information. We use the AutoAct subsets. For training, we use the synthetic and filtered agent trajectories released by AutoAct (Qiao et al., 2024). They are generated with LLaMA-2-70B-Chat. For evaluation, we use the test sets with three difficulty levels. They contain 300 questions for HotpotQA and 360 questions for ScienceQA. We report exact match (EM) after answer normalization on HotpotQA and accuracy on ScienceQA. We set the temperature to 0 and the seed to 42.

Adapter training. We follow the training setting of LRAgent (Jeon et al., 2026). We use the same training data and the same hyperparameters except for the learning rate. We train candidate adapters with different learning rates and select one adapter for each role on validation data that does not overlap with the test sets (Appendix D.2). We apply LoRA with rank $r = 8$ to the query and value projections. We set the LoRA scaling parameter α to 16. We absorb the factor α/r into B in the equations. Appendix F.3 reports the results with $r = 16$, 32 and 64 and with LoRA on the query, key, value and output projections. For a fair comparison, `BaseLRShared` and AIRMAS use the same plan adapter, which we train first following LRAgent (Jeon et al., 2026). `BaseLRShared` requires a single down-projection A for all agents. In our training protocol, we fix the A of this plan adapter and train only the up-projections B of the action and reflect adapters. AIRMAS trains the action and reflect adapters independently of the plan adapter with the original LoRA training procedure (Hu et al., 2022). The other schemes use the same adapters as AIRMAS. Appendix D.2 lists the hyperparameters and the training curves and describes the adapter selection. LRAgent reports that its shared-A adapters are more accurate than independently trained adapters. In our setting, the independently trained adapters are more accurate even without cache reuse (Appendix E). LRAgent reports the average of 20 evaluation runs, whereas we report a single greedy run.

Baselines. We compare AIRMAS with the cache-sharing schemes of Section 2.4 and with two KV reuse methods. `Non-Shared` keeps an independent KV cache for each role. When a role is invoked again, it reuses the longest common prefix of its own cache and prefills only the remaining tokens. `BaseShared` shares the base cache and computes the LR cache with the adapter of the current role. `BaseLRShared` (Jeon et al., 2026) also shares the LR cache, which requires a single A for all roles (Section 2.2). The other baselines reuse the full KV caches that other agents computed. `FullShared` reuses the full cache of the prefix without recomputation. CacheBlend (Yao et al., 2025) recomputes 15% of the reused tokens and reuses the cache of the rest. DroidSpeak (Liu et al., 2026) recomputes the KV cache of the layers that affect the accuracy most. Following LRAgent (Jeon et al., 2026), we recompute about one-third of the layers. The recomputation of a group

Table 2: Accuracy (%) on ScienceQA and exact match (%) on HotpotQA of cache-sharing schemes. `Non-Shared` reuses prefixes only within each role, and `BaseShared` recomputes the LR cache of the prefix with the current adapter. The schemes below the horizontal rule also reuse the LR cache or the full KV cache of other agents. Bold marks the best accuracy among these schemes.

	LLaMA-3.1-8B-Instruct		Ministral-8B-Instruct	
Method	ScienceQA	HotpotQA	ScienceQA	HotpotQA
`Non-Shared`	73.06	38.33	67.78	36.67
`BaseShared`	70.56	37.67	59.72	35.67
`FullShared`	61.67	30.00	44.44	28.33
CacheBlend + LoRA	63.61	30.67	50.00	29.67
DroidSpeak + LoRA	65.83	32.00	52.78	30.67
`BaseLRShared` (LRAgent)	65.28	34.67	54.72	34.33
AIRMAS	**66.39**	**37.33**	**58.06**	**35.00**

of consecutive layers starts from the hidden state at its first layer, which each agent stores for its own suffix tokens. We run both methods with the independently trained adapters and denote them CacheBlend + LoRA and DroidSpeak + LoRA. They operate on the full KV cache and do not split the value cache into a base cache and an LR cache. Appendix D.3 gives their configuration.

Efficiency protocol. The trajectory length varies with the question and with the outputs of the agents. Each scheme produces different outputs, so it can take a different number of steps and accumulate a different trajectory. We therefore compare the schemes under the same context length on the controlled trace of LRAgent (Jeon et al., 2026). It fixes the agent schedule and the prefill and generation lengths of every step (Appendix D.4). Only the retrieved context length varies from 0.25k to 16k tokens per retrieval step. This gives accumulated trajectory lengths L from 1.9k to 66.4k tokens. The trace thus also covers trajectories longer than the benchmark trajectories of 1.0k to 1.7k tokens (Appendix D.4). System throughput is the number of input and output tokens of the controlled trace, each counted once, divided by the model execution time. Its numerator is $912 + 4L_{ctx}$ for every scheme, where L_{ctx} is the retrieved context length (Appendix D.4). Total TTFT is the sum of the prefill latencies over all agent steps. `Non-Shared`, `FullShared`, CacheBlend + LoRA and DroidSpeak + LoRA use the scaled dot-product attention (SDPA) of PyTorch (Paszke et al., 2019). `BaseShared` and `BaseLRShared` use Flash-LoRA-Attention (Jeon et al., 2026). AIRMAS uses the fused kernel of Section 2.5 and invokes it through the same interface as `BaseLRShared`. All measurements run in FP16 on a single NVIDIA H100 NVL GPU.

3.2 Accuracy evaluation

Table 2 reports the accuracy of every scheme. `Non-Shared` and `BaseShared` do not reuse the LR cache of another agent, and we use them as accuracy references. Among the schemes below the horizontal rule, AIRMAS is the most accurate on both models and both benchmarks. It improves over `BaseLRShared` by 2.67% on HotpotQA with LLaMA-3.1-8B-Instruct and by 3.33% on ScienceQA with Ministral-8B-Instruct. AIRMAS keeps the independently trained A_i of every agent, whereas `BaseLRShared` requires a single A for all agents. With the shared-A adapters, `Non-Shared` also reaches 65.28% on ScienceQA with LLaMA-3.1-8B-Instruct, the same as `BaseLRShared` (Appendix E). Without cache reuse, the shared-A adapters are 7.78% below the independently trained adapters in this setting. In this setting, AIRMAS is 6.67% below `Non-Shared` with the same adapters. `FullShared` reuses the full cache although every adapter transforms the same tokens differently. It is up to 23.33% below `Non-Shared`. CacheBlend + LoRA and DroidSpeak + LoRA recover part of this drop by recomputing a subset of tokens or layers. CacheBlend + LoRA is less accurate than `BaseLRShared` in all four settings. DroidSpeak + LoRA is less accurate than `BaseLRShared` in three of them. AIRMAS remains below `BaseShared` in all four settings. The gap is at most 1.67% in three settings and 4.17% on ScienceQA with LLaMA-3.1-8B-Instruct. On HotpotQA with LLaMA-3.1-8B-Instruct, AIRMAS is 0.33% below `BaseShared` and 1.00% below `Non-Shared`. Across the four settings, AIRMAS recovers 21% to 89% of the gap between `BaseLRShared` and `BaseShared`.

Table 3: System throughput (tokens per second) of cache-sharing schemes on LLaMA-3.1-8B-Instruct at each total trajectory length of the controlled trace. Higher is better. SDPA is the scaled dot-product attention of PyTorch and Flash-LoRA is Flash-LoRA-Attention.

Method	Kernel	1.9k	3.0k	5.0k	9.1k	17.3k	33.7k	66.4k
`Non-Shared`	SDPA	256.0	383.0	606.2	1048.0	1641.0	2241.1	2260.8
`FullShared`	SDPA	277.4	408.3	670.2	1253.2	2115.8	3421.9	4270.0
CacheBlend + LoRA	SDPA	232.2	349.0	553.4	808.9	876.1	473.3	175.2
DroidSpeak + LoRA	SDPA	231.8	355.6	574.8	979.4	1671.0	2520.6	2780.1
`BaseShared`	Flash-LoRA	267.7	398.2	640.0	1062.5	1627.9	2047.7	1897.9
`BaseLRShared` (LRAgent)	Flash-LoRA	267.0	403.5	664.1	1180.2	2064.6	3249.8	4051.2
Ours (AIRMAS)	Fused	264.8	395.9	654.4	1158.2	1997.8	3261.6	4027.7

Table 4: Total TTFT (s) over the 17 agent steps of the controlled trace on LLaMA-3.1-8B-Instruct at each total trajectory length. Lower is better.

Method	Kernel	1.9k	3.0k	5.0k	9.1k	17.3k	33.7k	66.4k
`Non-Shared`	SDPA	0.53	0.61	0.90	1.56	3.26	7.85	22.19
`FullShared`	SDPA	0.42	0.47	0.55	0.71	1.22	2.69	7.83
CacheBlend + LoRA	SDPA	0.67	0.84	1.42	3.58	12.01	63.08	369.26
DroidSpeak + LoRA	SDPA	0.55	0.57	0.69	1.09	2.08	4.81	13.18
`BaseShared`	Flash-LoRA	0.50	0.60	0.89	1.62	3.62	9.50	28.15
`BaseLRShared` (LRAgent)	Flash-LoRA	0.43	0.45	0.55	0.79	1.43	3.29	9.38
Ours (AIRMAS)	Fused	0.44	0.45	0.55	0.80	1.45	3.30	9.47

3.3 Inference efficiency

Table 3 and Table 4 report the throughput and the total TTFT under the controlled trace. `Non-Shared` computes input tokens outside the longest common prefix with its own cache. This can include tokens previously processed by other roles. `BaseShared` recomputes the LR cache of the prefix with the current adapter. AIRMAS computes hidden states only for the L_{s_i} suffix tokens. At 66.4k tokens it reaches $1.78\times$ the throughput of `Non-Shared` and $2.12\times$ that of `BaseShared`. NonShared and `BaseShared` have up to $2.38\times$ and $2.97\times$ the total TTFT of AIRMAS. The comparison with `BaseLRShared` shows the cost of the transformation. Both schemes prefill the same L_{s_i} suffix input tokens. AIRMAS additionally transforms the stored suffix LR caches by $T_{i\to n-1}$. Its total TTFT stays within 2.4% of `BaseLRShared` at every trajectory length. Its throughput is $0.97\times$ to $1.00\times$ that of `BaseLRShared`. With a separate transformation step the throughput drops to $0.61\times$ to $0.78\times$ that of the fused kernel (Appendix C.2). CacheBlend + LoRA and DroidSpeak + LoRA recompute part of the KV cache at every agent step. Their throughput is below that of AIRMAS at every trajectory length. AIRMAS reaches $1.11\times$ to $1.45\times$ the throughput of DroidSpeak + LoRA. DroidSpeak + LoRA has up to $1.46\times$ the total TTFT of AIRMAS. The throughput of CacheBlend + LoRA decreases beyond 17.3k tokens. At 66.4k tokens, AIRMAS reaches $22.99\times$ the throughput of our CacheBlend + LoRA implementation. `FullShared` performs no adapter computation for the prefix. It has the lowest total TTFT at 1.9k tokens and from 9.1k tokens. It also has the highest throughput at every trajectory length. However, it is up to 23.33% less accurate than `Non-Shared` (Table 2). AIRMAS reaches $0.92\times$ to $0.98\times$ the throughput of `FullShared`. On HotpotQA and ScienceQA, AIRMAS has the same total TTFT as `BaseLRShared`, and its model latency is at most 0.08 s higher (Appendix F.1). Appendix F.2 reports the peak memory of every scheme.

4 Conclusion

We presented AIRMAS, an inference framework for sequential LoRA-based multi-agent systems. AIRMAS reuses the suffix LR caches of earlier agents with independently trained adapters. A lightweight $r \times r$ pseudo-inverse transformation approximates each prefix segment for the current agent, and a fused kernel applies the transformation within Flash-LoRA-Attention. The current agent computes hidden states only for its suffix tokens and keeps its own down-projection. On HotpotQA and ScienceQA with LLaMA-3.1-8B-Instruct and Ministral-8B-Instruct, AIRMAS improves accuracy over `BaseLRShared` with comparable inference time.

AI USE STATEMENT

We used ChatGPT and Codex (OpenAI) and Claude (Anthropic) as follows. We used them to implement parts of the methods and the evaluation code and to give feedback on the design of the experiments. They also helped us interpret experimental results. We used them to check the derivation in Appendix B. We used GPT-5.6 Terra (OpenAI) only to generate the reference trajectories of the first adapter-selection stage (Appendix D.2). We have not used generative AI tools to propose the method or to formulate hypotheses or mathematical claims. We have not used them to generate or clean the training or test data. Additionally, we used generative AI tools to draft parts of the paper, to edit the paper for readability, to search the literature and to format the references. The authors reviewed all AI-assisted work and tested all AI-generated code. Every reported number was checked against the measurement logs, and every reference was checked against its original publication. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work proposes an inference method for LoRA-based multi-agent systems. It does not involve human subjects or the collection of new data. We use the public HotpotQA and ScienceQA benchmarks, the agent trajectories released by AutoAct and publicly released models under their licenses. AIRMAS approximates the LR caches of the agents. Its outputs can therefore differ from those of the original adapters (Table 2). AIRMAS does not change the adapters or the training data, and it inherits the biases of the base models and the adapters. We are not aware of other ethical concerns of this work.

REPRODUCIBILITY STATEMENT

We submit the code of AIRMAS, the baselines and the evaluation as anonymous supplementary material. We also submit the trained adapters and the result logs. Because of a disclosure restriction, the code does not include the implementation of the external tools. The logs do not include the tool outputs. Exact reproduction of benchmark accuracy depends on the restricted tools and their responses. The efficiency measurements on the controlled trace depend only on the prefill and generation lengths of the steps (Appendix D.4). Section 2 and Appendix B describe the method and its cost. Appendix C.1 gives the fused kernel in Algorithm 1. Appendix D.2 lists the hyperparameters of adapter training and describes the adapter selection. Appendix D.3 gives the configuration of CacheBlend + LoRA and DroidSpeak + LoRA. Appendix D.4 describes the controlled trace and the metrics. Section 3.1 gives the decoding and hardware settings.

REFERENCES

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, and Sumit Sanghai. GQA: Training generalized multi-query transformer models from multi-head checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 4895–4901, 2023. doi: 10.18653/v1/2023.emnlp-main.298.

Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/98ed250b203d1ac6b24bbcf263e3d4a7-Abstract-Conference.html.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024. URL https://arxiv.org/abs/2407.21783v1.

Tianyu Fu, Zihan Min, Hanling Zhang, Jichao Yan, Guohao Dai, Wanli Ouyang, and Yu Wang. Cache-to-cache: Direct semantic communication between large language models. In *International Conference on Learning Representations (ICLR)*, 2026.

Yingsheng Geng, Yuchong Gao, Weihong Wu, Guyue Liu, and Jiang Liu. RelayCaching: Accelerating LLM collaboration via decoding KV cache reuse. In *International Conference on Machine Learning (ICML)*, 2026. URL https://arxiv.org/abs/2603.13289.

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 *International Conference on Learning Representations (ICLR)*, 2024.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations (ICLR)*, 2022.

Hyesung Jeon, Hyeongju Ha, and Jae-Joon Kim. LRAgent: Efficient KV cache sharing for multi-LoRA LLM agents. In *International Conference on Machine Learning (ICML)*, 2026.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. 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 (SOSP)*, pp. 611–626. ACM, 2023. doi: 10.1145/3600006.3613165.

Yuhan Liu, Yuyang Huang, Jiayi Yao, Shaoting Feng, Zhuohan Gu, Kuntai Du, Hanchen Li, Yihua Cheng, Junchen Jiang, Shan Lu, Madan Musuvathi, and Esha Choukse. DroidSpeak: KV cache sharing across fine-tuned model variants. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, Renton, WA, 2026. USENIX Association.

Pan Lu, Swaroop Mishra, Tanglin Xia, Liang Qiu, Kai-Wei Chang, Song-Chun Zhu, Oyvind Tafjord, Peter Clark, and Ashwin Kalyan. Learn to explain: Multimodal reasoning via thought chains for science question answering. In *Advances in Neural Information Processing Systems*, volume 35, pp. 2507–2521, 2022.

Mistral AI. Ministral-8B-Instruct-2410. Model card, 2024. URL https://huggingface.co/mistralai/Ministral-8B-Instruct-2410.

Zaifeng Pan, Ajjkumar Patel, Zhengding Hu, Yipeng Shen, Yue Guan, Wan-Lu Li, Lianhui Qin, Yida Wang, and Yufei Ding. KVFlow: Efficient prefix caching for accelerating LLM-based multi-agent workflows. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

Sangeun Park and Minhae Kwon. Multi^2: Hierarchical multi-agent decision-making with LLM-based agents in interactive environments. In *International Conference on Machine Learning (ICML)*, 2026.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32, pp. 8024–8035, 2019.

Shuofei Qiao, Ningyu Zhang, Runnan Fang, Yujie Luo, Wangchunshu Zhou, Yuchen Jiang, Chengfei Lv, and Huajun Chen. AutoAct: Automatic agent learning from scratch for QA via self-planning. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL), Volume 1: Long Papers*, pp. 3003–3021, Bangkok, Thailand, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.165.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *International Conference on Learning Representations (ICLR)*, 2024.

Weizhou Shen, Chenliang Li, Hongzhan Chen, Ming Yan, Xiaojun Quan, Hehong Chen, Ji Zhang, and Fei Huang. Small LLMs are weak tool learners: A multi-LLM agent. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 16658–16680, Miami, Florida, USA, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.929.

Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, Joseph E. Gonzalez, and Ion Stoica. S-LoRA: Serving thousands of concurrent LoRA adapters. In *Proceedings of Machine Learning and Systems (MLSys)*, 2024.

Xiangyu Shi, Marco Chiesa, Gerald Q. Maguire, Jr., and Dejan Kostić. KVComm: Enabling efficient LLM communication through selective KV sharing. In *International Conference on Learning Representations (ICLR)*, 2026. URL https://openreview.net/forum?id=F7rUng23nw.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. RoFormer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024. doi: 10.1016/j.neucom.2023.127063.

Yashar Talebirad and Amirhossein Nadiri. Multi-agent collaboration: Harnessing the power of intelligent LLM agents. *arXiv preprint arXiv:2306.03314*, 2023.

Chunlin Tian, Zhan Shi, Zhijiang Guo, Li Li, and Chengzhong Xu. HydraLoRA: An asymmetric LoRA architecture for efficient fine-tuning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Yiming Wang, Yu Lin, Xiaodong Zeng, and Guannan Zhang. MultiLoRA: Democratizing LoRA for better multi-task learning. *arXiv preprint arXiv:2311.11501*, 2023.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

Sunghyeon Woo, Jaeeun Kil, Hoseung Kim, Minsub Kim, Joonghoon Kim, Ahreum Seo, Sungjae Lee, Minjung Jo, Jiwon Ryu, Baeseong Park, Se Jung Kwon, and Dongsoo Lee. ICaRus: Identical cache reuse for efficient multi-model inference. In *International Conference on Learning Representations (ICLR)*, 2026.

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 *Conference on Language Modeling (COLM)*, 2024.

Yifei Xia, Fangcheng Fu, Wentao Zhang, Jiawei Jiang, and Bin Cui. Efficient multi-task LLM quantization and serving for multiple LoRA adapters. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Jingbo Yang, Bairu Hou, Wei Wei, Yujia Bao, and Shiyu Chang. KVLink: Accelerating large language models via efficient KV cache reuse. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025a.

Yaming Yang, Dilxat Muhtar, Yelong Shen, Yuefeng Zhan, Jianfeng Liu, Yujing Wang, Hao Sun, Weiwei Deng, Feng Sun, Qi Zhang, Weizhu Chen, and Yunhai Tong. MTL-LoRA: Low-rank adaptation for multi-task learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 22010–22018, 2025b.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 2369–2380, Brussels, Belgium, 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1259. URL https://aclanthology.org/D18-1259/.

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 (EuroSys)*. ACM, 2025. doi: 10.1145/3689031.3696098.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Hancheng Ye, Zhengqi Gao, Mingyuan Ma, Qinsi Wang, Yuzhe Fu, Ming-Yu Chung, Yueqian Lin, Zhijian Liu, Jianyi Zhang, Danyang Zhuo, and Yiran Chen. KVCOMM: Online cross-context KV-cache communication for efficient LLM-based multi-agent systems. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

Xiaoyan Yu, Tongxu Luo, Yifan Wei, Fangyu Lei, Yiming Huang, Hao Peng, and Liehuang Zhu. Neeko: Leveraging dynamic LoRA for efficient multi-character role-playing agent. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 12540–12557, Miami, Florida, USA, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.697.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Jianhui Zuo, Xuemeng Song, Haokun Wen, Meng Liu, Yupeng Hu, Jiuru Wang, and Liqiang Nie. D^{2}MoRA: Diversity-regulated asymmetric MoE-LoRA decomposition for efficient multi-task adaptation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 29286–29294, 2026.

A Related Work

A.1 LLM-Based Multi-Agent Systems

AutoGen (Wu et al., 2024) builds LLM applications with multiple agents that converse with each other to accomplish tasks. MetaGPT (Hong et al., 2024) encodes standardized operating procedures (SOPs) of human workflows into prompt sequences and assigns diverse roles to agents. Role-based fine-tuning provides an effective way to specialize agent behavior (Park & Kwon, 2026). For example, α-UMi (Shen et al., 2024) decomposes tool use into planner, caller, and summarizer models. AutoAct (Qiao et al., 2024) maintains a single backbone model and realizes each agent's role with its own LoRA adapter. Neeko (Yu et al., 2024) also uses LoRA to specialize an agent for distinct characters.

A.2 Multi-LoRA Adaptation and Cache Sharing

Multi-LoRA adaptation. Multi-LoRA systems attach multiple adapters to a shared frozen backbone for concurrent adapter serving (Sheng et al., 2024; Xia et al., 2024) and multi-task adaptation (Wang et al., 2023). One design choice is whether to share the down-projection matrix A across adapters. HydraLoRA (Tian et al., 2024) observes that down-projection parameters are similar across adapters trained on different subtasks. It therefore shares one A across multiple B matrices. Other multi-task LoRA methods also use shared down-projections (Yang et al., 2025b). LRAgent relies on this shared-A design for direct LR cache sharing (Jeon et al., 2026). It reports that its shared-A adapters are more accurate than independently trained adapters. In contrast, D^{2}MoRA (Zuo et al., 2026) identifies limitations of a single shared down-projection for expert specialization.

KV reuse across multiple agents with the same LLM. Prefix caching reuses the KV cache of an identical prefix within a single model (Kwon et al., 2023; Zheng et al., 2024). When shared content appears under different prefixes, its cache can differ because of changes in position and preceding context. CacheBlend (Yao et al., 2025) reuses cached non-prefix chunks and selectively

recomputes tokens with large KV deviations. KVFlow (Pan et al., 2025) uses the agent workflow to guide prefix-cache eviction and prefetching. KVLink (Yang et al., 2025a) and KVCOMM (Ye et al., 2025) address cache reuse across context segments or different prefixes. RelayCaching (Geng et al., 2026) reuses the decoding-phase KV caches of upstream agents and selectively recomputes tokens in the middle layers. These methods address reuse with the same model weights rather than differences between independently trained adapters.

KV reuse across different models. DroidSpeak (Liu et al., 2026) reuses KV caches between models fine-tuned from the same base model. It recomputes the caches of the critical layers found by profiling and reuses the caches of the remaining layers. It also stores the hidden states required by the first recomputed layer. Cache-to-Cache (Fu et al., 2026) trains a network to project and fuse a source model's KV cache into the target model's cache for semantic communication between models. KVComm of Shi et al. (2026), which is different from KVCOMM (Ye et al., 2025), shares the KV caches of selected layers between LLMs derived from the same base model. It selects the layers by attention importance. These methods operate on full-dimensional KV representations. DroidSpeak recomputes hidden states for previously processed context, and Cache-to-Cache requires training a cache-fusion module.

KV cache reuse across LoRA adapters. ICaRus (Woo et al., 2026) factors a decoder-only Transformer into a logical encoder that produces the KV cache and a logical decoder that predicts the next token. It freezes the logical encoder and fine-tunes only the logical decoder, so that task-specific models share an identical KV cache. ICaRus changes the fine-tuning procedure to obtain an exact cache reuse. AIRMAS keeps the existing adapters and approximates the LR cache of the target agent. LRAgent (Jeon et al., 2026) is the closest to our work. It decomposes the value cache as $F_i = X_i W_0 + (X_i A_i) B_i$ and stores $X_i A_i$ as an LR cache. Its Flash-LoRA-Attention applies attention to the LR cache before the up-projection (Section 2.5). Among the four caching schemes of LRAgent, only `BaseLRShared` shares the LR cache, and it requires a shared down-projection matrix.

B Detailed complexity analysis

Table 5 extends Table 1 with the weight and LoRA terms that the main text omits. `BaseShared` and `BaseLRShared` use Flash-LoRA-Attention as in LRAgent (Jeon et al., 2026). AIRMAS uses the fused kernel of Section 2.5.

Table 5: Detailed memory and computation of cache-sharing schemes up to constant factors. The notation follows Table 1 and M is the size of the shared backbone. The memory column includes the model weights, the stored caches and the transformation matrices.

		Computation	
Scheme	Memory	Prefill	Decode
`Non-Shared`	$M + Nrd_{in} + Nrd_{out} + NLd_{out}$	$Ld_{in}d_{out} + Ld_{in}r + Lrd_{out} + L^2 d_{out}$	$d_{in}d_{out} + d_{in}r + rd_{out} + Ld_{out}$
`BaseShared`	$M + Nrd_{in} + Nrd_{out} + Ld_{out} + NLr$	$Ld_{in}d_{out} + Ld_{in}r + Lrd_{out} + L^2 d_{out} + L^2 r$	$d_{in}d_{out} + d_{in}r + rd_{out} + Ld_{out} + Lr$
`BaseLRShared`	$M + rd_{in} + Nrd_{out} + Ld_{out} + Lr$	$L_{s_i} d_{in}d_{out} + L_{s_i} d_{in}r + L_{s_i} rd_{out} + L_{s_i} Ld_{out} + L_{s_i} Lr$	$d_{in}d_{out} + d_{in}r + rd_{out} + Ld_{out} + Lr$
Ours (AIRMAS)	$M + Nrd_{in} + Nrd_{out} + Ld_{out} + Lr + N^2 r^2$	$L_{s_i} d_{in}d_{out} + L_{s_i} d_{in}r + L_{s_i} rd_{out} + L_{s_i} Ld_{out} + L_{s_i} Lr + L_{s_i} Lr^2$	$d_{in}d_{out} + d_{in}r + rd_{out} + Ld_{out} + Lr + Lr^2$

Derivation. We count multiply-adds at one layer and sum them over the heads. We count the value projection with its LoRA terms and the attention over the value cache. The query and key projections scale in the same way as the value projection. We exclude the MLP layers, the output projection, the softmax and the offline precomputation of $T_{i \to j}$. The current agent i has at most $L = L_{p_i} + L_{s_i}$ tokens in its context, and every cache holds at most L tokens.

Memory. The backbone has M parameters. An adapter has rd_{in} parameters in A_i and rd_{out} parameters in B_i. N independently trained adapters add $Nrd_{in} + Nrd_{out}$. `BaseLRShared` shares one A and adds $rd_{in} + Nrd_{out}$. `Non-Shared` stores a full cache of Ld_{out} values for each agent, which gives NLd_{out}. `BaseShared` stores one base cache of Ld_{out} values and one LR cache of Lr values for each agent, which gives $Ld_{out} + NLr$. `BaseLRShared` stores one base cache and one LR cache, which gives $Ld_{out} + Lr$. AIRMAS stores one base cache and the suffix LR caches. The suffix lengths sum to L, so the suffix LR caches hold Lr values. The transformation matrices of all adapter pairs add $N(N-1)r^2 \leq N^2r^2$ values. AIRMAS also stores one byte per token for π_t (Appendix C.1), which is dominated by Lr.

Prefill. A scheme that recomputes the prefix processes all L tokens. The value projection costs $Ld_{in}d_{out}$. The LoRA terms cost $Ld_{in}r$ for XA and Lrd_{out} for the up-projection. Attention with L queries over L tokens costs L^2d_{out} for the value cache. This gives the prefill of `Non-Shared`. `BaseShared` runs the same forward pass. Flash-LoRA-Attention also applies the attention weight to the LR cache, which adds L^2r. `BaseLRShared` and AIRMAS process only the L_{s_i} suffix tokens. Their value projection and LoRA terms cost $L_{s_i}d_{in}d_{out}$, $L_{s_i}d_{in}r$ and $L_{s_i}rd_{out}$. Their L_{s_i} queries attend to all L tokens. This costs $L_{s_i}Ld_{out}$ for the base cache and $L_{s_i}Lr$ for the LR cache. AIRMAS also transforms the stored suffix LR caches of the earlier agents. The fused kernel transforms each loaded row once per query block (Appendix C.1). With query blocks of B_r rows this costs at most $\lceil L_{s_i}/B_r \rceil Lr^2 \leq L_{s_i}Lr^2$ for one head.

Decode. A decoding step processes one token. The value projection and the LoRA terms cost $d_{in}d_{out} + d_{in}r + rd_{out}$. The query attends to L tokens, which costs Ld_{out}. The schemes with an LR cache add Lr for the attention on the LR cache. AIRMAS transforms the loaded rows of the earlier agents again at every decoding step, which adds at most Lr^2.

Dominant terms. With $C = d_{in}d_{out} + Ld_{out}$ we have $LC = Ld_{in}d_{out} + L^2d_{out}$ and $L_{s_i}C = L_{s_i}d_{in}d_{out} + L_{s_i}Ld_{out}$. The prefill of `Non-Shared` is $\mathcal{O}(LC)$ in the worst case. The prefill of `BaseShared` is $\mathcal{O}(LC)$, and that of `BaseLRShared` is $\mathcal{O}(L_{s_i}C)$. Since $r \ll d_{in}, d_{out}$, the LoRA terms $Ld_{in}r$, Lrd_{out} and L^2r are dominated by LC, and their suffix counterparts by $L_{s_i}C$. The transformation term $L_{s_i}Lr^2$ is dominated by $L_{s_i}Ld_{out}$ only when $r^2 < d_{out}$. This holds for $r = 8$ and $d_{out} = 1024$ but not for $r = 64$ (Appendix F.3), so we keep it for AIRMAS. For the same reason the decode of AIRMAS is $\mathcal{O}(C + Lr^2)$, while the decode of every other scheme is $\mathcal{O}(C)$. These are the entries of Table 1.

C Flash-LoRA-Attention with on-the-fly transformation

C.1 Algorithm

Section 2.5 summarizes the fused kernel. This section gives its data layout, its steps and its cost.

Data layout. Let $\bar{Z}_{n-1} = [Z_0^*; \ldots; Z_{n-1}^*] \in \mathbb{R}^{L \times r}$ denote the stored suffix LR caches in token order before transformation. With repeated invocations, segments from the same agent need not be adjacent. Each row of $\bar{Z}_{n-1}$ corresponds to one token. An agent can be invoked several times in a trajectory (Appendix D.4). All its invocations use the same adapter, so the transformation depends only on the adapter. Let $\pi_t \in \{0, \ldots, N-1\}$ denote the agent whose adapter computed the stored LR-cache row for token t. Let A_{n-1} denote the down-projection of the current agent $n-1$. The subscript identifies the agent, not its invocation number. The row of token t is transformed by $T_{\pi_t \to n-1} = A_{\pi_t}^{\dagger} A_{n-1}$. Here A_{π_t} is the down-projection of adapter π_t. AIRMAS stores π_t with the suffix LR cache when token t is appended. This adds one byte per token. The $N(N-1)$ transformation matrices of every layer are stored in global memory. The kernel of agent $n-1$ reads the $N-1$ matrices $\{T_{k \to n-1}\}_{k \neq n-1}$ and the identity matrix $T_{n-1 \to n-1} = I$.

Two-kernel implementation. A separate transformation kernel reads the Lr values of $\bar{Z}_{n-1}$ from global memory. It writes the Lr values of $\widehat{Z}_{n-1}$ back to global memory. The attention kernel then reads the Lr values of $\widehat{Z}_{n-1}$ again. Each layer therefore moves $3Lr$ values of the LR cache through global memory at every decoding step and launches one more kernel. The fused kernel reads the Lr values of $\bar{Z}_{n-1}$ once and never writes $\widehat{Z}_{n-1}$.

Fused kernel. Algorithm 1 gives the forward pass for one head. It follows Flash-LoRA-Attention (Jeon et al., 2026) and adds lines 3 and 8. Each query block is processed by one thread block. The thread block first loads these N matrices into shared memory (line 3). They occupy Nr^2 values (768 bytes for $N = 3$ and $r = 8$ in FP32). The thread block then streams over the key/value blocks. For each block it loads the keys, the base cache and the stored suffix LR cache with its adapter indices (lines 6 and 7). The thread that loads a row $\bar{Z}_{n-1}[t, :]$ also reads π_t and selects $T_{\pi_t \to n-1}$. It computes

$$\widehat{Z}_{n-1}[t, :] = \bar{Z}_{n-1}[t, :]\, T_{\pi_t \to n-1} \tag{9}$$

in registers (line 8). The row has r values and the product takes r^2 multiply-adds. It accumulates the product in FP32 and rounds it once to FP16. Rows with $\pi_t = n - 1$ are already computed with A_{n-1}. They are multiplied by the identity matrix, so every row takes the same path and these rows keep their values. The transformed block $\widehat{Z}^{(j)}$ is written to shared memory next to the base cache block $P^{(j)}$. The block-wise attention weight $\tilde{S}$ is applied to both blocks (lines 12 and 13). The two accumulators share the online-softmax statistics $m^{(i)}$ and $\ell^{(i)}$. The LR accumulator $O_{lr}^{(i)}$ stays in rank r. After the last block, the kernel normalizes the LR accumulator and applies B_{n-1} once (line 16). A transformed block exists only in shared memory while its key/value block is processed.

Prefill and decoding. In prefill the queries are the $L_{s_{n-1}}$ suffix tokens. The causal mask lets each query attend to the prefix and to the earlier suffix tokens. Key/value blocks that lie entirely after a query block are fully masked and are skipped (line 5). In decoding there is a single query. The kernel packs the query heads that share a key/value head into one query block. It also splits the key/value blocks over several thread blocks. The kernel uses online softmax and parallel attention computation as in FlashAttention and FlashAttention-2 (Dao et al., 2022; Dao, 2024). Each split applies B_{n-1} to its LR accumulator before the partial outputs are combined. The LR row of each generated token is appended to Z^*_{n-1} with $\pi_t = n - 1$ and uses the identity matrix. The rows of the earlier agents are transformed again at every decoding step because $\widehat{Z}_{n-1}$ is not stored.

Correctness. The transformation is linear and is applied to each row before the attention weight. The block-wise attention weight $\tilde{S}$ and the online-softmax rescaling α are shared by $O^{(i)}$ and $O_{lr}^{(i)}$. After the last block $\mathrm{diag}(\ell^{(i)})^{-1} O_{lr}^{(i)}$ equals the rows of $S_{n-1}\widehat{Z}_{n-1}$ for query block i. The kernel therefore computes Eq. 8 with $\widehat{Z}_{n-1}$ up to floating-point rounding.

Overhead. A key/value block of B_c rows requires $B_c r^2$ multiply-adds for the transformation. For $r = 8$ this is 64 multiply-adds per row. Each thread block transforms the key/value blocks that it streams over. In prefill, the transformation is repeated for every query block and query head. In decoding, it is repeated for every key/value head and split. The additional memory is one byte per token for π_t and Nr^2 values of shared memory per thread block. The stored suffix LR caches and the transformation matrices remain unchanged.

C.2 Kernel ablation

Configurations. The fused kernel in Section 2.5 applies the transformation within Flash-LoRA-Attention and removes the separate transformation kernel. Table 6 compares three configurations under the controlled trace of Appendix D.4. `BaseLRShared` with Flash-LoRA calls Flash-LoRA-Attention without the transformation. AIRMAS with Separate + Flash-LoRA transforms the stored suffix LR caches in its own step. It writes the transformed LR cache to a temporary buffer in global memory and then calls Flash-LoRA-Attention. AIRMAS with Fused uses the fused kernel of Section 2.5. All three process the same tokens. `BaseLRShared` and Fused invoke the attention kernel through the same interface. The difference between Fused and `BaseLRShared` is the cost of the transformation. The difference between Separate + Flash-LoRA and Fused is the overhead of our separate implementation.

Results. The transformation adds minimal overhead in the fused kernel. The TBT of Fused stays within 3.9% of `BaseLRShared` at every trajectory length, for example 25.18 against 25.13 ms at 66.4k tokens. Its total TTFT stays within 2.4% of `BaseLRShared`. Separate + Flash-LoRA

Algorithm 1 Flash-LoRA-Attention with on-the-fly transformation (one head)

Require: $L = L_{p_{n-1}} + L_{s_{n-1}}$, $Q \in \mathbb{R}^{L_{s_{n-1}} \times d_{head}}$, $K \in \mathbb{R}^{L \times d_{head}}$, base cache $P_{n-1} \in \mathbb{R}^{L \times d_{head}}$
Require: stored suffix LR caches $\bar{Z}_{n-1} \in \mathbb{R}^{L \times r}$ in token order, adapter index $\pi \in \{0, \ldots, N-1\}^L$
Require: transformation matrices $\{T_{k \to n-1}\}_{k=0}^{N-1}$ with $T_{n-1 \to n-1} = I$, up-projection $B_{n-1} \in \mathbb{R}^{r \times d_{head}}$
Require: query block size B_r, key/value block size B_c
Ensure: attention output $O \in \mathbb{R}^{L_{s_{n-1}} \times d_{head}}$

Split Q and O into blocks $Q^{(i)}, O^{(i)}$ of B_r rows. Split K, P_{n-1}, $\bar{Z}_{n-1}$ and π into blocks $K^{(j)}, P^{(j)}, \bar{Z}^{(j)}, \pi^{(j)}$ of B_c rows.
for each query block i **in parallel** (one thread block) **do**
  Load $\{T_{k \to n-1}\}_{k=0}^{N-1}$ to shared memory. {added}
  Load $Q^{(i)}$ to shared memory. Initialize $O^{(i)} \leftarrow 0 \in \mathbb{R}^{B_r \times d_{head}}$, $O_{lr}^{(i)} \leftarrow 0 \in \mathbb{R}^{B_r \times r}$, $m^{(i)} \leftarrow -\infty$, $\ell^{(i)} \leftarrow 0$.
  for each key/value block j that is not fully masked for query block i **do**
    Load $K^{(j)}$ and $P^{(j)}$ to shared memory.
    Load $\bar{Z}^{(j)}$ and $\pi^{(j)}$.
    For each row t: $\widehat{Z}^{(j)}[t,:] \leftarrow \bar{Z}^{(j)}[t,:]\, T_{\pi_t \to n-1}$ in FP32. Write $\widehat{Z}^{(j)}$ to shared memory. {added}
    $E \leftarrow \mathrm{Mask}\big(Q^{(i)} K^{(j)\top} / \sqrt{d_{head}}\big)$
    $m^{new} \leftarrow \max\big(m^{(i)}, \mathrm{rowmax}(E)\big)$, $\alpha \leftarrow \exp(m^{(i)} - m^{new})$, $\tilde{S} \leftarrow \exp(E - m^{new})$
    $\ell^{(i)} \leftarrow \alpha \odot \ell^{(i)} + \mathrm{rowsum}(\tilde{S})$
    $O^{(i)} \leftarrow \mathrm{diag}(\alpha)\, O^{(i)} + \tilde{S} P^{(j)}$
    $O_{lr}^{(i)} \leftarrow \mathrm{diag}(\alpha)\, O_{lr}^{(i)} + \tilde{S} \widehat{Z}^{(j)}$
    $m^{(i)} \leftarrow m^{new}$
  end for
  $O^{(i)} \leftarrow \mathrm{diag}(\ell^{(i)})^{-1} O^{(i)} + \big(\mathrm{diag}(\ell^{(i)})^{-1} O_{lr}^{(i)}\big) B_{n-1}$. Write $O^{(i)}$ to global memory.
end for

Table 6: Kernel ablation under the controlled trace on LLaMA-3.1-8B-Instruct. Flash-LoRA is Flash-LoRA-Attention. Separate + Flash-LoRA transforms the stored suffix LR caches in its own step and then calls Flash-LoRA-Attention. Fused is the fused kernel of AIRMAS. TBT is the average time between two generated tokens. TTFT is the sum over the 17 agent steps.

Method	Kernel	1.9k	3.0k	5.0k	9.1k	17.3k	33.7k	66.4k
TBT (ms) ↓								
`BaseLRShared` (LRAgent)	Flash-LoRA	24.44	24.67	25.04	24.79	24.89	25.34	25.13
AIRMAS	Separate + Flash-LoRA	39.15	39.83	40.41	40.40	39.97	39.96	40.37
Ours (AIRMAS)	Fused	24.62	25.16	25.43	25.30	25.84	25.16	25.18
TTFT (s) ↓								
`BaseLRShared` (LRAgent)	Flash-LoRA	0.43	0.45	0.55	0.79	1.43	3.29	9.38
AIRMAS	Separate + Flash-LoRA	0.88	1.14	1.16	1.35	1.99	3.84	9.92
Ours (AIRMAS)	Fused	0.44	0.45	0.55	0.80	1.45	3.30	9.47
Throughput (tokens/s) ↑								
`BaseLRShared` (LRAgent)	Flash-LoRA	267.0	403.5	664.1	1180.2	2064.6	3249.8	4051.2
AIRMAS	Separate + Flash-LoRA	164.0	241.5	402.6	721.4	1316.0	2246.2	3136.1
Ours (AIRMAS)	Fused	264.8	395.9	654.4	1158.2	1997.8	3261.6	4027.7

adds 14.1 to 15.2 ms per decoding step relative to Fused. Its TBT is 1.55 to 1.60× that of Fused. It also adds 0.44 to 0.69 s to the TTFT, which is 26 to 41 ms for each agent step. The throughput of Separate + Flash-LoRA is therefore only 0.61 to 0.78× that of Fused. This overhead does not increase with the trajectory length, although the total trajectory length increases 34× from 1.9k to 66.4k tokens. The controlled trace fixes the agent schedule. Every agent step transforms the same number of segments at every trajectory length. Only the retrieved context length varies. At 66.4k tokens $\widehat{Z}_{n-1}$ is only 1.0 MiB at each layer. The overhead is therefore not explained by the size of the

LR cache. The TBT of `BaseLRShared` is 24.44 ms at 1.9k tokens and 25.13 ms at 66.4k tokens. The TBT in our experiments is thus dominated by costs that do not increase with the trajectory length.

Lightweight launch path. Our implementation also provides a launch path with a lower CPU cost for the fused kernel. It reuses preallocated buffers and the transformation matrices across calls. It reads Q, K and V directly in the layout of the decoder layer through their strides. In an isolated benchmark, it reduces the CPU time of one call from 22.5 to 6.5 μs. We do not use this path in Section 3.3, so that AIRMAS and `BaseLRShared` invoke the attention kernel through the same interface.

D Implementation details

D.1 AutoAct workflow and trajectory templates

We follow the AutoAct (Qiao et al., 2024) setup of LRAgent (Jeon et al., 2026). A trajectory accumulates a system prompt, the user question and multiple rounds of agent outputs. Tool outputs retrieved from external sources are inserted between the agent outputs. The system prompt specifies the thought, action and observation format. It is identical for all agents and is therefore fully shared. Previously processed tokens keep their positions. KVLink (Yang et al., 2025a) and KVCOMM (Ye et al., 2025) correct the deviation of a reused cache under a different prefix. With identical prefixes they reuse the cache directly, so they reduce to `FullShared` in this setup (Jeon et al., 2026). KVFlow (Pan et al., 2025) changes only the eviction and prefetching of prefix caches.

The agents run in a fixed order of plan, plan and action. The plan agent first writes a thought. In its second step it selects a tool. The action agent then generates the arguments of the selected tool. When the tool retrieves a context, the retrieved context is appended to the trajectory as an observation. When the selected tool is Finish, the action agent outputs a final answer. The reflect agent then decides in two steps whether the answer is sufficient. It first writes a thought and then outputs its decision. It can override an incorrect Finish and return control to the plan agent. The number of agent iterations is limited to 15 per question.

D.2 Adapter training

The training data follow Section 3.1. Table 7 lists the selected learning rates and the other hyperparameters. We first train the plan adapter and use it in both `BaseLRShared` and AIRMAS. For `BaseLRShared` we fix the A of this plan adapter and train only the B of the action and reflect adapters. For AIRMAS we train the action and reflect adapters independently. The other schemes use the adapters of AIRMAS. LRAgent evaluates every scheme with its shared-A adapters. We evaluate every scheme except `BaseLRShared` with the independently trained adapters. The accuracy drops in Table 2 are therefore not directly comparable with those reported by LRAgent. Figure 4 shows the training loss and the gradient norm of the three adapters on ScienceQA with LLaMA-3.1-8B-Instruct. The loss of every adapter drops sharply in the first two epochs and changes little after the fourth epoch. The action adapter uses the largest learning rate, and its gradient norm is the largest through most of the training. It also starts from the highest loss, and the reflect adapter has the lowest loss after the fourth epoch. The action adapter starts from the highest loss in every training run of Figures 4 to 6.

Selected learning rates. No single learning rate is selected for every role (Table 7). On ScienceQA with LLaMA-3.1-8B-Instruct, the selected learning rates range from 1×10^{-5} for the plan agent to 2×10^{-4} for the action agent. With Ministral-8B-Instruct, the action agent instead receives the smallest learning rate of 1×10^{-5}. The selected learning rates also change with the rank and the adapted projections (Figures 5 and 6). These results support selecting the learning rate separately for each role.

Adapter selection. For each model and benchmark, we train candidate adapters with the learning rates in $\{1 \times 10^{-5}, 2 \times 10^{-5}, 4 \times 10^{-5}, 5 \times 10^{-5}, 7 \times 10^{-5}, 9 \times 10^{-5}, 1 \times 10^{-4}, 2 \times 10^{-4}\}$. We select one adapter for each role in two stages on validation data that does not overlap with the test

sets. We first select the plan adapter with the first stage. The independently trained adapters and the shared-A adapters use this plan adapter. We then train the action and reflect candidates on the same training data. The independently trained candidates learn both A and B. The shared-A candidates reuse the A of the selected plan adapter and learn only B. For each of the two adapter sets, we fix the plan adapter and select the action and reflect adapters with both stages.

In the first stage, we score each candidate on its own role with teacher forcing. We fix the reference trajectory before the step of the role, and the candidate generates only the next output of that role. This stage uses 36 ScienceQA questions with 36, 52 and 72 plan, action and reflect steps, and 30 HotpotQA questions with 30, 116 and 60 steps. We generate these trajectories with GPT-5.6 Terra (OpenAI). A plan output is correct if its first tool exactly matches the reference. We score an action output by the input that it passes to the tool, not by the tool name, which the plan agent selects. For Search, Retrieve and Lookup, the score is the F1 score between this input and the reference input. A reflect output is correct if its right or wrong decision matches the reference. We keep the best plan adapter and the top three action and reflect adapters, including ties at the boundary.

In the second stage, we fix the selected plan adapter and run full episodes of the three agents with every remaining pair of action and reflect adapters. The episodes use `Non-Shared` with the candidate adapters, a reference attention implementation and greedy decoding. They are limited to 10 agent steps and 128 generated tokens per step. For each benchmark, we use three disjoint sets of 30 questions with 10 questions at each difficulty level. We draw them from the development split of HotpotQA and the test split of ScienceQA and exclude the questions of our test sets. We run all N combinations on the first set and keep the top $\lceil 2N/3 \rceil$ and their ties. After the second set, we keep the top $\lceil N/3 \rceil$ and their ties. We rank the combinations by the total number of correct answers and then by the smallest number of correct answers over the sets. After the third set, we select the combination with the most correct answers over the 90 questions. We break ties by the smallest number of correct answers over the sets and then by the candidate index. We exclude runs with missing rows, model or tool errors or mismatched input hashes from the ranking. Because we select the adapters with `Non-Shared`, the selection does not depend on any cache-sharing scheme.

Table 7: Hyperparameters of adapter training. The learning rates are given for the plan, action and reflect agents.

Hyperparameter	LLaMA-3.1-8B-Instruct	Ministral-8B-Instruct
Optimizer	AdamW	
Batch size	32	
Learning rate scheduler	cosine	
Epochs	10	
Warmup ratio	0.05	
Weight decay	0	
Adapted projections	query, value	
Rank r	8	
LoRA α	16	
LoRA dropout	0.05	
Max sequence length	32768	
Learning rate on ScienceQA	1×10^{-5} / 2×10^{-4} / 2×10^{-5}	5×10^{-5} / 1×10^{-5} / 2×10^{-5}

D.3 Configuration of CacheBlend and DroidSpeak

CacheBlend + LoRA. CacheBlend (Yao et al., 2025) recomputes the KV cache of the tokens whose reused cache deviates most from the recomputed one. We recompute 15% of the reused tokens without calibration. At layer 0, the current agent computes the key and value of every token with its own adapter. For each reused token, we sum the squared difference between the stored $[K; V]$ and the recomputed $[K; V]$ at layer 0 over all heads and dimensions. We select the 15% of the reused tokens with the largest difference, rounded up, and break ties by the earlier position. The selected tokens and all new tokens form the recomputed set. CacheBlend refines the selection over layers (Yao et al., 2025). We simplify this and keep the set of layer 0 for all layers. From layer 1, the current agent computes the projections, the attention and the MLP only for the recomputed set. The queries of the recomputed set attend to the keys and values of all positions. At each layer, the KV cache takes the reused cache and replaces the entries of the recomputed set with the recomputed

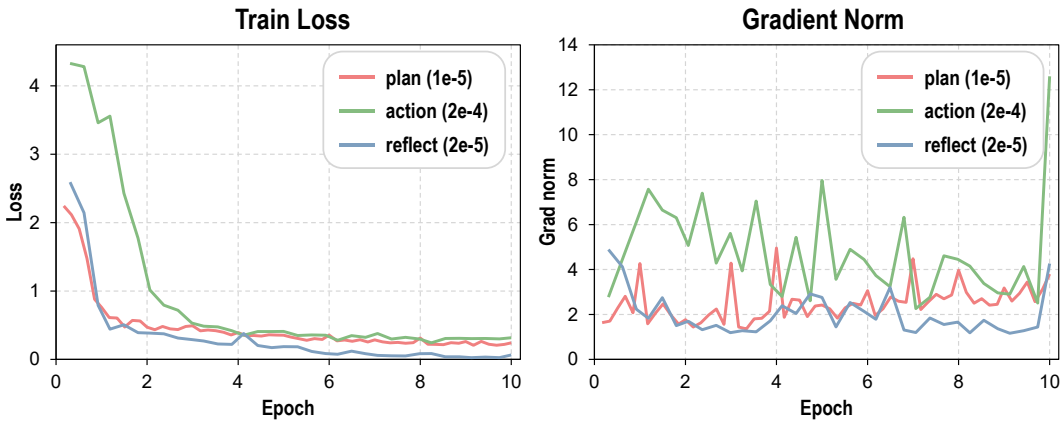

Figure 4: Training loss and gradient norm of the three role-specific adapters on ScienceQA with LLaMA-3.1-8B-Instruct. LoRA is applied to the query and value projections with $r = 8$. The legend gives the learning rate of each agent.

ones. The current agent stores this blended cache as its own cache. When it is invoked again, only the tokens added since its previous invocation are candidates for recomputation. Each role therefore keeps its own cache, as in `Non-Shared`. CacheBlend + LoRA stores no hidden state beyond the KV cache.

DroidSpeak + LoRA. DroidSpeak (Liu et al., 2026) recomputes the KV cache of the layers that affect the accuracy most when the cache of another agent is reused. Following LRAgent (Jeon et al., 2026), we recompute approximately one-third of the layers. This gives 11 of the 32 layers of LLaMA-3.1-8B-Instruct and 12 of the 36 layers of Ministral-8B-Instruct. We select the layers on a calibration set of 50 ScienceQA questions with 16, 17 and 17 questions from the three grade groups. The calibration set does not overlap with the training data, the 360 test questions or the validation sets. For each layer ℓ, we run full episodes on the calibration set. The agents reuse the KV cache of the other agents only at layer ℓ and recompute all other layers. We measure the accuracy drop from `Non-Shared` and select the layers with the largest drop. We break ties by the lower agreement of the output tokens with `Non-Shared` and then by the lower layer index. This single-layer probing simplifies the group-wise profiling of DroidSpeak. Table 8 lists the selected layers, and we use them on both benchmarks. Consecutive selected layers form a group. The current agent recomputes the KV cache of each group with its own adapter. The recomputation of a group starts from the hidden state at its first layer. DroidSpeak stores this hidden state as the E cache. Each agent stores the E cache only for its own suffix tokens. For the prefix, the current agent receives the E cache of each segment from the agent that processed the segment. It concatenates these E caches in token order and recomputes the KV cache of each group from them. This is the same layout as the stored suffix LR caches of AIRMAS (Appendix C.1). Under grouped-query attention (GQA) (Ainslie et al., 2023), the E cache of a layer is larger than its KV cache (Jeon et al., 2026). It is the extra memory of DroidSpeak + LoRA in Appendix F.2.

Selected layers. The selected layers differ between the two models (Table 8). For LLaMA-3.1-8B-Instruct, 9 of the 11 selected layers lie in the first half of the model. For Ministral-8B-Instruct, 10 of the 12 selected layers lie in the second half. The critical layers thus depend on the model. The consecutive selected layers form 7 groups for LLaMA-3.1-8B-Instruct and 4 groups for Ministral-8B-Instruct. Each group needs one E cache. LLaMA-3.1-8B-Instruct thus stores 7 hidden states of 4096 values for every token. This is 44% of the 65,536 key and value entries that its KV cache holds for one token.

D.4 Controlled trace and trajectory lengths

The trajectory length varies with the question and with the outputs of the agents (Section 3.1). We therefore compare efficiency under the same context length on the controlled trace of LRAgent (Jeon

Table 8: Layers recomputed by DroidSpeak + LoRA. They are approximately one-third of the layers with the largest accuracy drop in the single-layer probe on the ScienceQA calibration set. Layers are indexed from 0.

Model	Selected layers
LLaMA-3.1-8B-Instruct (11 of 32)	0, 1, 5, 6, 9, 10, 12, 13, 15, 19, 27
Ministral-8B-Instruct (12 of 36)	15, 16, 22, 24, 25, 26, 27, 28, 29, 31, 32, 33

Table 9: Controlled trace of one run with the prefill and generation lengths of each step (Jeon et al., 2026). L_{ctx} is the retrieved context length.

Step	Agent	Prefill	Generation	Step	Agent	Prefill	Generation
1	plan	512	32	10	plan	L_{ctx}	32
2	plan	8	8	11	plan	8	8
3	action	8	8	12	action	8	8
4	plan	L_{ctx}	32	13	plan	L_{ctx}	32
5	plan	8	8	14	plan	8	8
6	action	8	8	15	action	8	8
7	plan	L_{ctx}	32	16	reflect	32	32
8	plan	8	8	17	reflect	8	8
9	action	8	8				

et al., 2026). It is built from profiled trajectories. It fixes the agent schedule and the prefill and generation lengths of every step (Table 9). One controlled-trace run contains 17 agent steps. It consists of five plan-plan-action cycles and two reflect steps. Only the retrieved context length L_{ctx} varies. It is prefilled at the first plan step of the second to the fifth cycle. The prefill lengths sum to $632 + 4L_{ctx}$ and the generation lengths sum to 280. The total trajectory length of a controlled-trace run is therefore $912 + 4L_{ctx}$. We set $L_{ctx} \in \{256, 512, 1024, 2048, 4096, 8192, 16384\}$ tokens, corresponding to the context settings 0.25k through 16k in LRAgent. This gives total trajectory lengths of 1.9k, 3.0k, 5.0k, 9.1k, 17.3k, 33.7k and 66.4k tokens. Total lengths are rounded to one decimal place in thousands of tokens. They are the columns of Table 3 and Table 4. We do not use $L_{ctx} = 32$k. It gives 132.0k tokens, which exceeds the maximum context length of 128k tokens of LLaMA-3.1-8B-Instruct. Apart from the first step and the retrieval steps every step prefills 8 to 32 tokens. These suffix lengths are on the order of 10^1 as in Table 1. The four retrieval steps account for most of the prefill. They prefill 1,024 of the 1,656 prefill tokens (61.8%) at 1.9k tokens and 65,536 of the 66,168 prefill tokens (99.0%) at 66.4k tokens. The other 13 steps prefill 632 tokens in total, and the 17 steps generate 280 tokens at every trajectory length. A longer trajectory therefore changes only the prefill of the retrieval steps and the length of the attended context.

Metrics. System throughput counts each input and output token of the 17 agent steps once:

$$\text{Throughput} = \frac{(632 + 4L_{ctx}) + 280}{T_{\text{model}}} = \frac{912 + 4L_{ctx}}{T_{\text{model}}} \quad \text{tokens/s.} \tag{10}$$

Here, T_{model} is the wall-clock time of the full model trace. Recomputed prefix tokens are not counted again in the numerator, which is identical for all schemes. The denominator includes prefill and decoding. Every agent step prefills its input before it generates its first token. TTFT is the prefill latency of an agent step. We report its sum over the 17 agent steps as the total TTFT, as in LRAgent. TBT is the average time between two consecutive generated tokens.

Benchmark trajectories. Table 10 reports the suffix lengths and the trajectory lengths of the trajectories that `Non-Shared` generates on HotpotQA and ScienceQA. The plan agent has the longest suffix because it prefills the first step and the retrieved contexts. The action and reflect agents prefill 16.0 to 45.0 tokens on average. These suffix lengths are on the order of 10^1 as in the controlled trace. The final trajectory has 1047.1 to 1659.3 tokens. HotpotQA gives longer trajectories than ScienceQA with both models. They are $1.46\times$ longer with LLaMA-3.1-8B-Instruct and $1.58\times$ longer with Ministral-8B-Instruct. The average suffix of the action and reflect agents is 7% to 19% of that of the plan agent. The context of an action or reflect step thus consists mostly of prefix tokens, whose hidden states AIRMAS does not recompute.

Table 10: Average suffix length of each agent and average trajectory length (tokens) of the trajectories that `Non-Shared` generates on the 360 questions of ScienceQA and the 300 questions of HotpotQA. Trajectory is the final accumulated trajectory length.

		Suffix length			
Model	Benchmark	Plan	Action	Reflect	Trajectory
LLaMA-3.1-8B-Instruct	ScienceQA	235.1	45.0	16.0	1074.2
	HotpotQA	212.5	32.4	18.1	1567.3
Ministral-8B-Instruct	ScienceQA	218.4	28.6	18.0	1047.1
	HotpotQA	225.5	32.1	20.3	1659.3

E ADAPTER MARGIN ATTRIBUTION

Section 3.2 compares the accuracy of the schemes. This section examines how the adapter of the current agent affects the correct answer under each scheme. Let z denote the logits of an answer token and y the correct token. The margin of the correct answer is

$$m = z_y - \max_{y' \neq y} z_{y'}. \tag{11}$$

We scale the adapter update of the current agent from $\Delta W = AB$ to $g\Delta W$ in every layer and adapted projection. For the reused LR cache, this scales the up-projection B that the current agent applies to it. We define the adapter margin attribution (AMA) as the change of the margin with the adapter scale,

$$\mathrm{AMA} = \left.\frac{\partial m}{\partial g}\right|_{g=1}. \tag{12}$$

A positive AMA means that strengthening the adapter moves the model toward the correct answer. An AMA near zero means that the margin changes little when the adapter scale changes. The fused kernel has no backward pass, so we estimate AMA by the central difference $\big(m(1+\epsilon)-m(1-\epsilon)\big)/2\epsilon$ with $\epsilon = 0.05$. With $\epsilon = 0.1$, the mean AMA of every setting changes by at most 0.03. On 20 questions with `Non-Shared`, the estimate agrees in sign with backpropagation in every case. All settings replay the same fixed trajectory that `Non-Shared` generates with the independently trained adapters. We teacher-force the gold answer in the final agent step and compute the margin of the gold answer token. The caches of the earlier agents stay fixed, and only the final step is recomputed for each value of g. We use the 325 of the 360 ScienceQA questions whose trajectory ends with a well-formed answer within the step limit and has an unambiguous answer span.

Table 11: Adapter margin attribution (AMA) on ScienceQA with LLaMA-3.1-8B-Instruct over the 325 eligible questions. The AMA of a single question can be negative. Positive is the percentage of questions with a positive AMA. Accuracy is measured over all 360 questions with the protocol of Table 2.

Method	Accuracy (%)	Mean AMA	Positive (%)
Independently trained A_i			
Non-Shared	73.06	0.79	56.3
Ours (AIRMAS)	66.39	2.60	75.4
Shared A			
Non-Shared	65.28	1.00	64.6
BaseLRShared (LRAgent)	65.28	0.90	60.3

Table 11 reports the results. With the shared-A adapters, `Non-Shared` and `BaseLRShared` reach the same accuracy and a similar mean AMA. Under a shared A, cache reuse changes little in the effect of the adapter. With the independently trained adapters, `Non-Shared` has a small mean AMA of 0.79, and 56.3% of the questions have a positive AMA. Its margin changes little when the adapter scale changes. AIRMAS has a mean AMA of 2.60, and 75.4% of the questions have a positive AMA. The 95% bootstrap intervals of the two means do not overlap. With the transformed caches, the adapter of the current agent still moves the margin toward the correct answer. Its effect, however, has not saturated as it has under full recomputation. This is consistent with a

weaker adapter contribution under the transformed caches. A larger mean AMA does not imply a higher accuracy. `Non-Shared` with the independently trained adapters is more accurate than with the shared-A adapters, but its mean AMA is smaller (0.79 against 1.00). AMA therefore compares the effect of the adapter between schemes that use the same adapters. Cache reuse changes the mean AMA in opposite directions for the two adapter families. With the shared-A adapters, `BaseLRShared` lowers it by 0.10 and the share of positive AMA by 4.3%. With the independently trained adapters, AIRMAS raises it by 1.81 and the share of positive AMA by 19.1%.

F Additional experiments

F.1 Model latency on HotpotQA and ScienceQA

Section 3.3 feeds the same controlled trace to every scheme. On the benchmarks, each scheme generates its own trajectory. Its length varies with the question and with the outputs of the agents (Section 3.1). This section measures the latency on these trajectories with LLaMA-3.1-8B-Instruct. Model latency is the sum of the forward times over the agent steps of a question. It includes prefill, decoding and the forward pass that commits the last generated token to the cache. It excludes the time of the external tools. Total TTFT and TBT follow the definitions of Appendix D.4. We average the model latency and the total TTFT over the questions. The number of processed tokens differs between schemes, so we do not report system throughput on the benchmarks. The model latency and the total TTFT also depend on the number of agent steps, which differs between schemes.

Table 12: Model latency (s), total TTFT (s) and TBT (ms) of cache-sharing schemes on ScienceQA and HotpotQA with LLaMA-3.1-8B-Instruct. Model latency and total TTFT are averaged over the questions. Total TTFT is the sum over the agent steps of a question.

	ScienceQA			HotpotQA		
Method	Model	TTFT	TBT	Model	TTFT	TBT
Non-Shared	3.28	0.24	24.18	6.83	0.58	24.49
BaseShared	3.13	0.22	23.02	6.31	0.52	22.69
FullShared	3.50	0.23	25.91	7.13	0.57	25.74
CacheBlend + LoRA	3.55	0.28	25.95	7.48	0.74	26.43
DroidSpeak + LoRA	3.79	0.25	28.12	7.88	0.62	28.44
BaseLRShared (LRAgent)	3.10	0.20	23.01	6.35	0.48	23.05
Ours (AIRMAS)	3.18	0.20	23.66	6.37	0.48	23.10

Table 12 reports the results. The trajectories of `Non-Shared` have 1.0k to 1.7k tokens on average (Table 10). They are shorter than the shortest trajectory of the controlled trace (1.9k tokens). At 1.9k tokens of the controlled trace, the throughput of the schemes differs by at most $1.20\times$, against $24.37\times$ at 66.4k tokens (Table 3). AIRMAS has the same total TTFT as `BaseLRShared` on both benchmarks. Its model latency is 0.08 s higher on ScienceQA and 0.02 s higher on HotpotQA. Its TBT is 0.65 ms and 0.05 ms higher, respectively. Compared with `Non-Shared`, AIRMAS reduces the total TTFT by 0.04 s on ScienceQA and by 0.10 s on HotpotQA. Its model latency is 0.10 s and 0.46 s lower, respectively. `BaseShared` has a higher total TTFT than AIRMAS by 0.02 s and 0.04 s. Its TBT is lower, and its model latency is 0.05 s and 0.06 s lower than that of AIRMAS. On both benchmarks, the schemes that use Flash-LoRA-Attention or the fused kernel have a lower TBT than the schemes that use SDPA. `FullShared`, CacheBlend + LoRA and DroidSpeak + LoRA have a higher model latency than `Non-Shared` on both benchmarks. DroidSpeak + LoRA has the highest model latency and the highest TBT. CacheBlend + LoRA has the highest total TTFT. On both benchmarks, the schemes rank in the same order by model latency and by TBT. The total TTFT is only 6% to 10% of the model latency for every scheme. The model latency on the benchmarks thus follows the decoding time more than the prefill time. The model latency of AIRMAS is 6.7% lower than that of `Non-Shared` on HotpotQA and 3.0% lower on ScienceQA. HotpotQA gives longer trajectories (Table 10). The controlled trace also shows a larger gain over `Non-Shared` at longer trajectories (Table 3).

F.2 MEMORY USAGE

Table 1 gives the memory complexity of each scheme. We measure the memory usage on LLaMA-3.1-8B-Instruct under the controlled trace of Appendix D.4. Table 13 reports the peak memory at each trajectory length.

Definitions. Peak memory is the largest GPU memory allocated during the run. It includes the model weights of 14.98 GiB and temporary buffers. KV memory is every stored key and value cache. For `BaseShared`, `BaseLRShared` and AIRMAS it consists of the base key cache, the base value cache and the LR cache. Extra memory is the state that a scheme keeps beyond its KV cache. It is the E cache for DroidSpeak + LoRA (Appendix D.3) and the transformation matrices for AIRMAS. CacheBlend + LoRA stores no hidden state. Extra memory excludes the 1.5 MiB hidden-state buffer that every scheme holds, the 16 MiB attention workspace and the 1.5 MiB scaled up-projection shared by `BaseShared`, `BaseLRShared` and AIRMAS. Peak memory includes all of them.

Table 13: Peak GPU memory (GiB) of cache-sharing schemes under the controlled trace on LLaMA-3.1-8B-Instruct at each total trajectory length. It includes the model weights of 14.98 GiB.

Method	1.9k	3.0k	5.0k	9.1k	17.3k	33.7k	66.4k
Non-Shared	16.00	16.78	18.34	19.81	23.81	31.81	47.81
FullShared	15.26	15.51	16.01	16.29	17.29	19.29	23.29
CacheBlend + LoRA	15.96	16.46	17.48	19.50	23.57	31.69	47.94
DroidSpeak + LoRA	16.32	17.05	18.56	21.22	26.78	37.88	60.09
BaseShared	15.28	15.54	16.04	16.33	17.34	19.38	23.46
BaseLRShared (LRAgent)	15.28	15.53	16.03	16.31	17.32	19.32	23.35
Ours (AIRMAS)	15.28	15.53	16.03	16.31	17.32	19.32	23.35

Fixed trajectory lengths. Table 13 shows how the peak memory increases with the trajectory length. From 1.9k to 66.4k tokens, the peak memory of `FullShared`, `BaseShared`, `BaseLRShared` and AIRMAS increases by 8.03 to 8.18 GiB. These schemes keep a single shared cache and stay between 23.29 and 23.46 GiB at 66.4k tokens. `Non-Shared` keeps a separate full cache for each agent. Its peak memory increases by 31.81 GiB and reaches 47.81 GiB at 66.4k tokens. CacheBlend + LoRA recomputes a subset of tokens at every layer with the adapter of the current agent. Its peak memory of 47.94 GiB at 66.4k tokens is close to that of `Non-Shared`. DroidSpeak + LoRA has the largest peak memory at every trajectory length and reaches 60.09 GiB at 66.4k tokens. In addition to its KV caches, it stores the E cache for recomputation (Appendix D.3). `BaseShared` keeps one LR cache for each adapter. At 66.4k tokens, its peak memory is 0.11 GiB higher than that of `BaseLRShared`. AIRMAS stores the same base cache and suffix LR caches as `BaseLRShared`. Its peak memory matches that of `BaseLRShared` to two decimals at every trajectory length. Its extra memory is 0.047 MiB at both 17.3k and 66.4k tokens (Table 15). It consists of the transformation matrices, which do not grow with the trajectory length. AIRMAS never materializes $\widehat{Z}_{n-1}$ in global memory, so it adds no trajectory-dependent buffer for $\widehat{Z}$; its only additional storage is the fixed transformation matrices. At 66.4k tokens, AIRMAS uses 24.46 GiB less peak memory than `Non-Shared` and 36.74 GiB less than DroidSpeak + LoRA. At 1.9k tokens, its difference from `Non-Shared` is 0.72 GiB. The memory saving of AIRMAS thus grows with the trajectory length.

F.3 RANK AND ADAPTED PROJECTIONS

The main experiments use $r = 8$ (Section 3.1). $Z_i^* T_{i\to j} = X_i^* A_i A_i^\dagger A_j$ keeps only the component of X_i^* in the r-dimensional column space of A_i (Section 2.3). A larger r increases the maximum dimension of this subspace, but this does not guarantee a smaller approximation error for separately trained adapters. A larger r also increases the Lr^2 transformation cost (Table 1). We train the adapters with $r = 16$, 32 and 64 and evaluate AIRMAS with each rank. `Non-Shared` uses the same adapters and gives the reference accuracy at each rank. `BaseLRShared` uses the adapters with a shared A at the same rank, which are trained as in Section 3.1. Table 14 compares the

accuracy of AIRMAS and `BaseLRShared`. Table 15 reports the KV memory, the LR cache and the extra memory of AIRMAS at each rank. The LR cache grows with r and the transformation matrices grow with r^2 (Table 1). Figure 5 shows the training loss and the gradient norm at each rank. The legend gives the learning rate of each agent.

Accuracy across ranks. AIRMAS is more accurate than `BaseLRShared` at $r = 8$, 32 and 64 by 1.11%, 1.94% and 1.39% (Table 14). At $r = 16$, it is 0.28% below `BaseLRShared`, which is one question. The accuracy of `Non-Shared` does not change monotonically with the rank. At $r = 64$, both `BaseLRShared` and AIRMAS are more accurate than `Non-Shared`. The selected learning rates differ between the ranks (Figure 5), so Table 14 does not isolate the effect of the rank.

Memory across ranks. The LR cache doubles with each doubling of r (Table 15). At 66.4k tokens, it grows from 33.1 MiB at $r = 8$ to 265 MiB at $r = 64$. The extra memory grows by $4\times$ with each doubling of r and does not change with the trajectory length. At $r = 64$ and 66.4k tokens, the LR cache and the extra memory together are about 3% of the KV memory.

Training curves across ranks. The action adapter starts from the highest loss at every rank (Figure 5). At $r = 64$, the reflect adapter uses the smallest learning rate of 1×10^{-5}, and its loss decreases the most slowly.

Table 14: Accuracy (%) of AIRMAS and `BaseLRShared` with different LoRA ranks r on ScienceQA with LLaMA-3.1-8B-Instruct. `Non-Shared` uses the same adapters as AIRMAS. `BaseLRShared` uses the adapters with a shared A at the same rank.

	Accuracy (%) ↑		
r	Non-Shared	BaseLRShared (LRAgent)	Ours (AIRMAS)
8	73.06	65.28	66.39
16	66.94	65.00	64.72
32	71.11	65.83	67.78
64	66.39	68.89	70.28

Table 15: KV memory (GiB), LR cache (MiB) and extra memory (MiB) of AIRMAS with different LoRA ranks r under the controlled trace on LLaMA-3.1-8B-Instruct. KV memory includes the LR cache. Extra memory is the transformation matrices.

	17.3k tokens			66.4k tokens		
r	KV (GiB)	LR (MiB)	Extra (MiB)	KV (GiB)	LR (MiB)	Extra (MiB)
8	2.35	9.13	0.047	8.51	33.1	0.047
16	2.36	18.3	0.188	8.54	66.3	0.188
32	2.38	36.5	0.750	8.61	133	0.750
64	2.41	73.0	3.00	8.74	265	3.00

Adapted projections. The main experiments apply LoRA to the query and value projections as in LRAgent (Jeon et al., 2026). We also apply LoRA to the query, key, value and output projections with $r = 8$. LoRA on the key projection adds an LR cache to the key cache. AIRMAS transforms only the LR cache of the value projection and reuses the key cache of the earlier agents without a transformation. Table 16 reports the average cosine similarity between the caches of two agents over 128 questions of ScienceQA. Both agents process the same question without the agent prompts and the tool outputs. For each token and each of the 32 layers we concatenate the 8 key/value heads and compute the cosine similarity. We average it over the tokens, the layers and then over the questions. The key cache after RoPE (Su et al., 2024) has a similarity of 0.995 to 0.997 with LoRA on the query and value projections. It has 0.994 to 0.996 with LoRA on the query, key, value and output projections. The value cache has a lower similarity of 0.973 to 0.984 in both settings. From QV to QKVO, the key similarity changes by at most 0.0023, while the value similarity changes by up to 0.0085. LRAgent reports a similarity above 0.98 for the key cache with LoRA on the query and value projections (Jeon et al., 2026). The two settings train different adapters, so this comparison

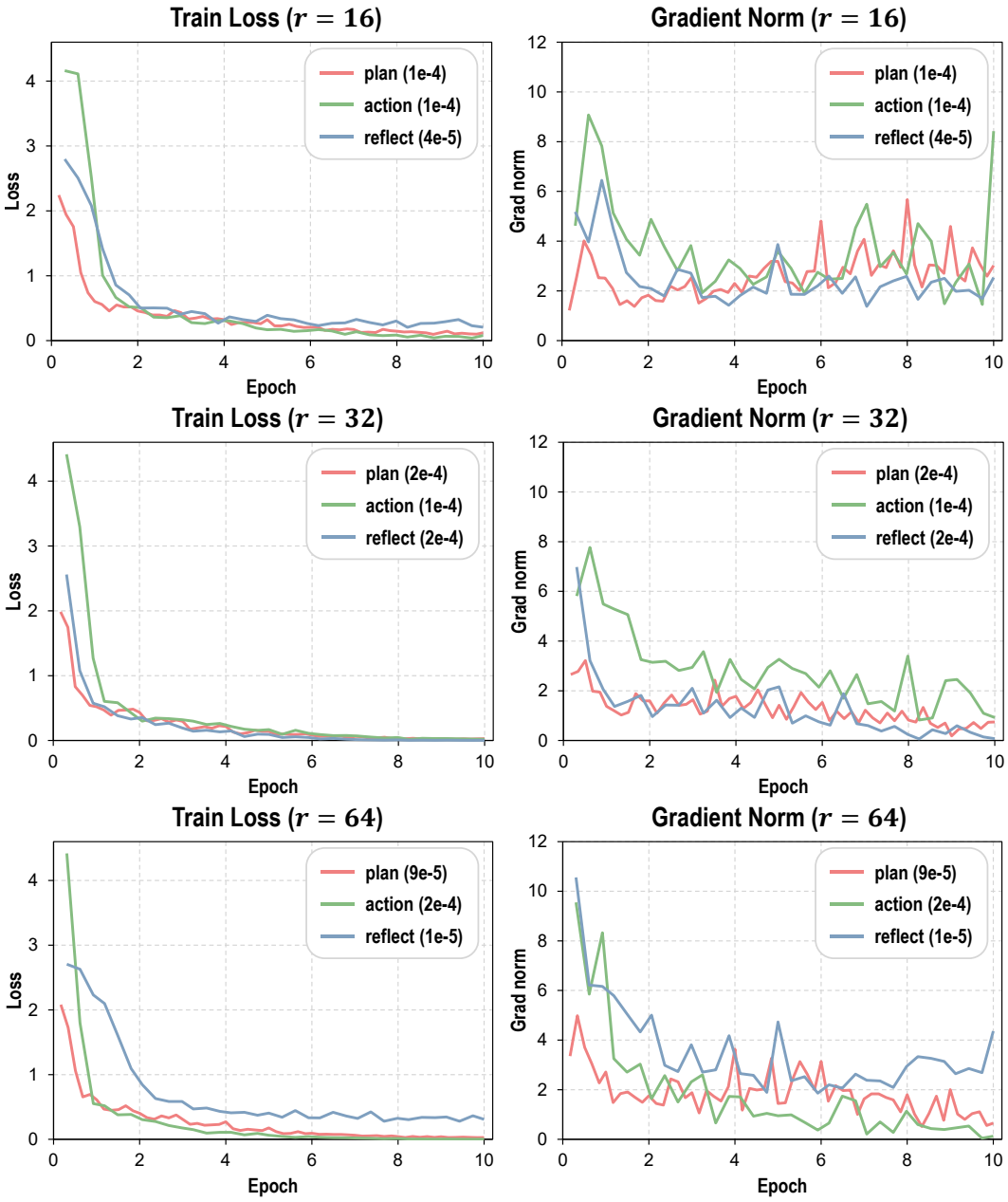

Figure 5: Training loss and gradient norm of the three role-specific adapters on ScienceQA with LLaMA-3.1-8B-Instruct. LoRA is applied to the query and value projections with $r = 16$, 32 and 64 from top to bottom. The legend gives the learning rate of each agent.

does not isolate the effect of LoRA on the key projection. The accuracy in Table 17 is therefore the main evidence for reusing the key cache. We compare `Non-Shared` and AIRMAS on ScienceQA with LLaMA-3.1-8B-Instruct. Table 17 reports the accuracy. `Non-Shared` reaches 66.67 with LoRA on the query, key, value and output projections, against 73.06 with LoRA on the query and value projections. AIRMAS reaches 70.28, which is above `Non-Shared` with the same adapters. Figure 6 shows the training loss and the gradient norm of the three adapters in this setting. The selected learning rates with QKVO differ from those with QV (Figures 4 and 6). Table 17 thus compares adapters that differ in both the adapted projections and the learning rates.

Table 16: Average cosine similarity of the key cache and the value cache between two agents over 128 questions of ScienceQA with LLaMA-3.1-8B-Instruct and $r = 8$. QV denotes LoRA on the query and value projections. QKVO denotes LoRA on the query, key, value and output projections. The key cache is taken after RoPE.

	Key cache		Value cache	
Agent pair	QV	QKVO	QV	QKVO
Plan–Action	0.9966	0.9943	0.9815	0.9730
Plan–Reflect	0.9950	0.9949	0.9741	0.9754
Action–Reflect	0.9955	0.9963	0.9757	0.9837

Table 17: Accuracy (%) with LoRA on different projections on ScienceQA with LLaMA-3.1-8B-Instruct and $r = 8$. `Non-Shared` uses the same adapters as AIRMAS. With LoRA on the key projection AIRMAS reuses the key cache without a transformation.

Adapted projections	Non-Shared	Ours (AIRMAS)
QV	73.06	66.39
QKVO	66.67	70.28

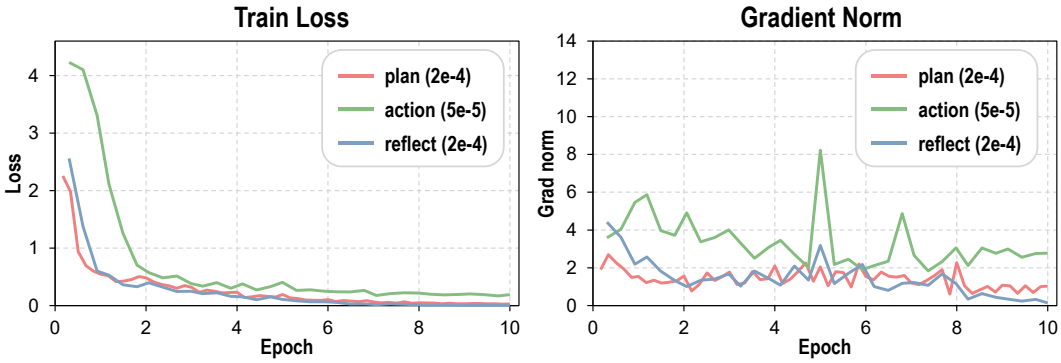

Figure 6: Training loss and gradient norm of the three role-specific adapters on ScienceQA with LLaMA-3.1-8B-Instruct. LoRA is applied to the query, key, value and output projections with $r = 8$. The legend gives the learning rate of each agent.