ATTUNEKV: RECEIVER-AWARE KV-CACHE HANDOFF FOR EFFICIENT MULTI-AGENT INFERENCE

Anonymous authors
Paper under double-blind review

ABSTRACT

Multi-agent large language model (LLM) systems frequently process messages generated by other agents. When agents share a model, the sender's key–value (KV) cache can be transferred to avoid redundant prefilling, but the imported cache is context-dependent and may not match the receiver's context. Moreover, the receiver may need only part of the message during decoding. KV handoff therefore poses two receiver-side resource decisions: where to spend recomputation and which transferred entries to keep in memory. We present AttuneKV, a receiver-aware KV-cache handoff method that obtains both decisions from one lightweight receiver observation. AttuneKV forwards one existing receiver-side token over the imported cache. Its layerwise attention ranks message positions for recomputation, while the attention-output change induced by removing each entry ranks positions for retention. The same observation supports both decisions without forwarding the incoming message positions for selection or maintaining a persistent KV reference pool. At 12,288 incoming tokens, AttuneKV achieves a 2.92× receiver time to first token (TTFT) speedup over dense prefill and a 1.45× speedup over RelayCaching. It also reduces KV-handoff memory by 2.0× and 34× relative to dense prefill and KVCOMM, respectively. Across math problem solving, broad-domain knowledge evaluation, and code generation, AttuneKV remains within 3.92 pp of dense prefill performance.

1 INTRODUCTION

Multi-agent large language model (LLM) systems solve complex tasks by allowing specialized agents to exchange natural-language messages and build on one another's outputs (Wu et al., 2023; Hong et al., 2024; Qian et al., 2024). This communication also creates substantial repeated computation. An upstream agent's response is often inserted into the prompt of one or more downstream agents, and every receiver must process the incoming message before generating its own response. During prefill, the LLM computes the key and value representations used by self-attention and stores them in the key–value (KV) cache for subsequent decoding (Kwon et al., 2023). When the sender and receiver share the same model, however, the KV states of the transferred message already exist from the sender's generation. Transferring these states therefore creates a natural opportunity to avoid repeatedly prefilling the same message at every receiver.

The challenge is that KV states are context-dependent. The representation of a token depends on the context preceding it, while different agents typically have different role instructions, conversation histories, and previously received messages. The sender's KV cache therefore differs from the cache that the receiver would obtain by prefilling the same message under its own context. Directly importing the sender's KV can consequently change the receiver's predictions. Recent systems address this mismatch through selective recomputation or context-dependent KV correction (Yao et al., 2025; Ye et al., 2025; Geng et al., 2026).

One source of overhead is finding which tokens need recomputation. Selective recomputation can update only a small fraction of the incoming message, but obtaining that fraction may itself require processing many message positions. CacheBlend begins by recomputing all message tokens in an early layer and progressively narrows the recomputing token set in later layers (Yao et al., 2025). RelayCaching skips shallow layers and confines rectification to a critical middle-layer range, recomputing all message positions up to a detection layer before selecting positions for sparse

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.

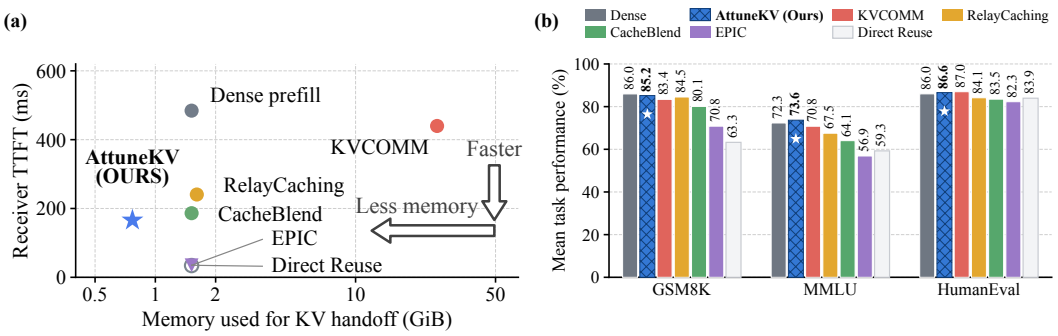

Figure 1: **AttuneKV reduces receiver latency and KV-cache memory while preserving task quality.** (a) Receiver time to first token (TTFT) versus memory used for KV handoff at 12,288 incoming tokens. Lower is better on both axes. (b) Mean task performance on GSM8K, MMLU, and HumanEval across the evaluated communication workflows. Higher is better.

correction (Geng et al., 2026). As messages become longer, a receiver may therefore recompute only a small subset of tokens while still spending substantial time identifying that subset. Efficient selective recomputation depends not only on how many tokens are recomputed, but also on how cheaply those tokens can be identified.

A second source of overhead is memory. KV reuse already consumes GPU memory because transferred message KV is kept for subsequent decoding. KVCOMM reduces receiver-side recomputation by maintaining a persistent KV reference pool that estimates context-dependent KV corrections, allowing transferred KV to be corrected without full contextual prefill (Ye et al., 2025). This reduces recomputation, but adds memory overhead. In a fully connected system with N agents, there are $N(N-1)$ sender-receiver pairs, and maintaining KV pools across these pairs can make pool memory scale as $O(N^2)$. Together, transferred message KV and the KV pool leave less GPU memory for concurrent agents and longer contexts. Efficient KV handoff should therefore reduce both receiver-side computation and memory use.

These two overheads lead to two design questions for receiver-side KV reuse. First, which transferred KV positions are worth recomputing, and how can the receiver identify them without another message-wide pass? Second, which transferred KV entries are worth retaining, so that memory use can be reduced while preserving task quality?

We present **AttuneKV**, a receiver-aware KV-cache handoff method that addresses both questions using a single receiver-side token forward. After the transferred KV is placed in the receiver's context, AttuneKV forwards one receiver-owned suffix token already present in the formatted prompt. This token produces one query at every layer that attends over the preceding cache. Consequently, even when the incoming messages contain thousands of tokens, the receiver obtains its selection signals by forwarding one query row per layer rather than forwarding the message positions themselves.

AttuneKV uses this receiver-side information in two ways. Receiver attention prioritizes message positions for recomputation, while the attention-output change associated with removing an entry guides retention. The two decisions therefore use signals obtained from the same token forward, while recomputation and retention remain independently configurable. Section 2 explains why these signals are suitable for the two decisions, and Section 3 describes how AttuneKV uses them during KV handoff.

Figure 1 previews the resulting system trade-off. AttuneKV reduces receiver latency and KV-cache memory while maintaining task quality close to dense prefill across the evaluated workloads.

At the longest evaluated efficiency input of 12,288 incoming tokens, AttuneKV achieves a $2.92\times$ receiver-TTFT speedup over dense prefill and a $1.45\times$ speedup over RelayCaching. At the same operating point, it retains approximately half of the transferred message KV. Across the evaluated benchmark–workflow settings, the largest absolute task-quality difference between AttuneKV and dense prefill is 3.92 percentage points.

These results motivate the central design of AttuneKV: acquire receiver-side information cheaply, then use separate signals to decide where recomputation is worth spending and which transferred KV entries are worth retaining.

Our contributions are threefold.

- We introduce a one-token receiver-side forward whose attention signal prioritizes transferred KV positions for recomputation without re-prefilling the incoming message.
- We use the same forward to obtain a separate removal-based signal for KV retention, allowing receiver-side computation and KV-cache memory to be controlled independently without maintaining a persistent KV reference pool.
- We evaluate AttuneKV across three benchmarks and three multi-agent communication workflows, measuring receiver latency and KV-cache memory together with task quality.

2 DESIGN RATIONALE

Section 1 identifies two decisions over transferred KV. The receiver must decide which positions are worth recomputing and which entries are worth retaining. Let T denote the receiver-owned suffix positions that follow the incoming messages in the formatted receiver prompt, and let $t_0 = \min T$ denote the first suffix token.[1] Because t_0 follows the incoming messages, it can attend to the transferred KV together with the preceding receiver context. AttuneKV uses this token to obtain the signals for both decisions. Figure 2 places this design in the broader space of KV reuse methods.

Both decisions can be examined through the attention read produced by t_0. For the local analysis, consider one attention head of dimension d_h and let $q \in \mathbb{R}^{d_h}$ denote the query produced by t_0. Let $\mathcal{V}$ denote the positions visible to this query, with keys k_i and values v_i. The attention weight on position i and the resulting head output are

$$a_i = \mathrm{softmax}_{j\in\mathcal{V}}\left(\frac{q^\top k_j}{\sqrt{d_h}}\right)_i, \qquad o = \sum_{i\in\mathcal{V}} a_i v_i. \tag{1}$$

Receiver attention indicates where recomputation is worth spending, while the attention-output change caused by removing an entry indicates which transferred KV is worth retaining.

2.1 RECEIVER ATTENTION FOR RECOMPUTATION

Suppose recomputing position i changed its cached representation by $(\Delta k_i, \Delta v_i)$. Holding the query and the other cache rows fixed, the corresponding first-order change in the attention output is

$$u_i^{\mathrm{rec}} \simeq a_i\left[\Delta v_i + \frac{q^\top \Delta k_i}{\sqrt{d_h}}(v_i - o)\right]. \tag{2}$$

This expression follows the attention-output sensitivity used in prior KV-reuse analysis (Yang et al., 2025). The attention weight a_i scales both the value correction and the effect of the key correction. A position receiving little receiver attention therefore has limited first-order influence on this attention read, even if its stored KV differs from the representation produced under dense receiver prefill. Receiver attention provides a signal that is both context-dependent and already observable. AttuneKV therefore uses attention as a sensitivity signal for allocating recomputation rather than attempting to estimate the unavailable correction itself.

The remaining question is whether one suffix token t_0 provides a useful ranking. We examine 512 fixed GSM8K receiver events in which the final decision agent reads upstream agent responses and produces the final answer. For each event, the problem, incoming messages, and receiver input are held fixed. Specifically, we compare the recomputed positions selected through the first suffix token, full suffix, random choice, and oracle (with dense prefill) in terms of later attention coverage, which

[1] In the chat templates used by our models, t_0 closes the current user turn: <|eot_id|> for Llama-3.1-Instruct and <|im_end|> for Qwen2.5-Instruct. AttuneKV reuses the first token following the final transferred message in the original formatted receiver prompt rather than inserting an additional token.

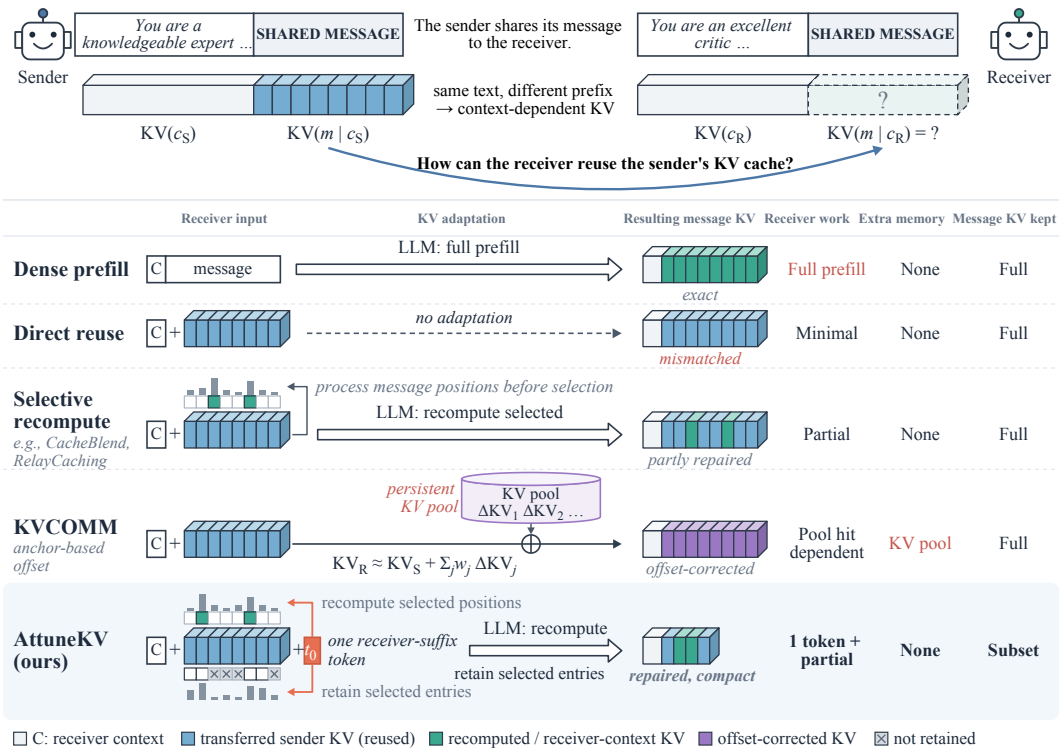

Figure 2: **Design space for transferred KV reuse.** Existing approaches either recompute the full receiver input, directly reuse sender KV, process message positions before selective correction, or use a persistent KV pool for context-dependent correction. AttuneKV instead uses one existing receiver-suffix token to guide both recomputation and retention, reducing receiver-side computation while keeping only selected transferred KV.

is the fraction of later receiver attention to incoming messages that falls on the selected positions. More details are provided in Appendix A.1.

At a 10% selection budget, the median later attention coverage is 42.5% for recomputed positions selected by the first suffix token, compared with 42.6% when using the complete suffix, 54.9% for the oracle, and 10.0% for random selection (Figure 3(a)). The first suffix token therefore achieves nearly the same median coverage as the complete suffix at this operating point. Appendix A.2 further compares the forward cost and prediction divergence of the two choices.

2.2 ATTENTION-OUTPUT CHANGE FOR RETENTION

Retention requires a different signal because removing an entry acts on attention differently from updating its representation. Consider removing position i while keeping the query and every surviving key and value fixed. The remaining attention weights are renormalized. For $a_i < 1$,

$$o_{-i} = \frac{o - a_i v_i}{1 - a_i}, \qquad u_i^{\text{drop}} = o_{-i} - o = -\frac{a_i}{1 - a_i}(v_i - o), \qquad r_i = \frac{a_i}{1 - a_i}\|v_i - o\|_2. \quad (3)$$

The last quantity r_i is the local removal sensitivity: the exact magnitude of the single-row removal effect for this attention read. Here, the factor a_i measures how strongly the receiver reads the entry, $v_i - o$ measures how distinct its value contribution is from the current head output, and $1/(1 - a_i)$ accounts for renormalization after removal. Importantly, all three quantities are already available from the same receiver observation used for recomputation scoring and from the transferred cache itself. Retention therefore requires no second message-wide forward and no persistent KV reference pool. AttuneKV uses r_i to rank transferred KV entries for retention. Section 3.2 describes how the per-head scores are aggregated across layers and heads.

To test how much transferred KV can be removed using this ranking before receiver predictions change substantially, we fix the recomputed cache and vary only the retained message KV. We evaluate the same 512 fixed GSM8K receiver events and measure the additional 32-step Jensen–Shannon divergence (JSD) relative to full retention under the same output sequence.

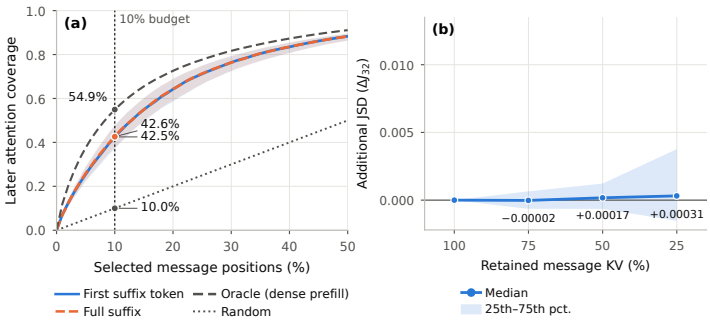

Figure 3: **Controlled diagnostics for the two receiver-side decisions.** (a) Later attention coverage. Lines show medians across 512 events; shading indicates the interquartile range. (b) ΔJ_{32}: change in 32-step JSD relative to full retention of the same recomputed cache.

A lower retention ratio results in more memory storage saving and larger spread of additional JSD, which is a fundamental tradeoff. As shown in Figure 3(b), a retention ratio of 50% retains approximately half of the message KV with only moderate spread of additional JSD. Therefore, we adopt the retention ratio of 50% as the default operating point hereafter.

3 AttuneKV

Section 2 motivates separate signals for recomputation and retention. AttuneKV aggregates these signals across layers and heads, then uses them to select which transferred KV positions to recompute and which entries to retain. Figure 4 summarizes the complete handoff.

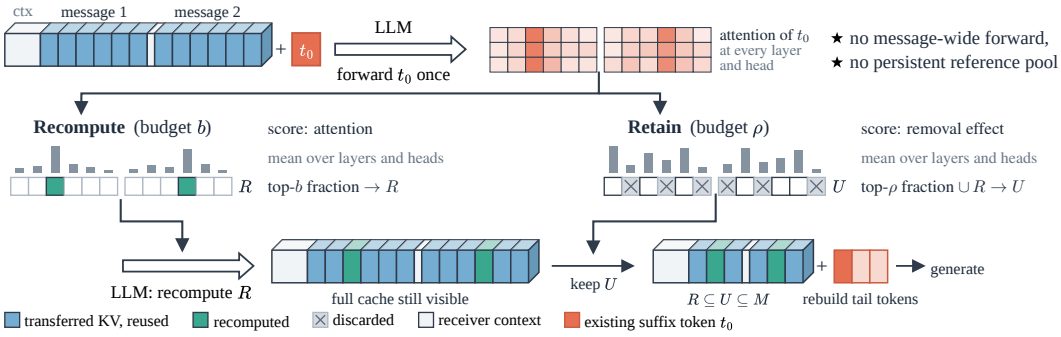

Figure 4: **AttuneKV algorithm overview.** One original receiver-suffix token provides the attention and removal-effect signals used for recomputation and retention. AttuneKV recomputes R while the full message cache remains visible, retains U, and rebuilds the original suffix T before generation.

3.1 Receiver-Suffix Token Forward

We consider senders and a receiver that share a model checkpoint and tokenizer. Let M denote the union of incoming-message positions in the receiver prompt, excluding receiver-owned context and separators. Using the notation from Section 2, AttuneKV forwards t_0, the first token in the receiver-owned suffix T, to obtain the two receiver-side signals.

Before selection, the receiver prepares its own prefix and intervening prompt tokens and imports the sender-generated message KV. Imported keys are remapped from their sender positions to their receiver positions using the model's rotary position schedule. Values require no positional transformation. Cache assembly and RoPE position remapping are described in Appendix C.

AttuneKV forwards only t_0 through the model using this assembled cache. At layer ℓ and query head h, let $q_{t_0}^{\ell h} \in \mathbb{R}^{d_h}$ denote its projected and position-encoded query. For grouped-query attention, let

$g(h)$ map query head h to the KV head that supplies its keys and values. This forward produces

$$a_i^{\ell h} = \mathrm{softmax}_{j \in \mathcal{V}_{t_0}} \left(\frac{\left\langle q_{t_0}^{\ell h}, k_j^{\ell, g(h)} \right\rangle}{\sqrt{d_h}} \right)_i, \qquad o^{\ell h} = \sum_{j \in \mathcal{V}_{t_0}} a_j^{\ell h} v_j^{\ell, g(h)}, \tag{4}$$

where $\mathcal{V}_{t_0}$ is the complete causal domain visible to t_0. The softmax therefore includes receiver-owned context, transferred messages, separators, and t_0 itself. Selection is subsequently restricted to positions in M.

This one forward supplies all quantities required by both decisions. Recomputation uses the receiver attention $a_i^{\ell h}$. For retention, the single-head removal sensitivity from Equation (3) becomes

$$r_i^{\ell h} = \frac{a_i^{\ell h}}{1 - a_i^{\ell h}} \left\| v_i^{\ell, g(h)} - o^{\ell h} \right\|_2. \tag{5}$$

AttuneKV therefore obtains both signals from the same token forward and does not perform a second message-wide forward for retention selection.

3.2 Allocating Computation and Memory

AttuneKV aggregates the layerwise quantities into separate token-level scores for recomputation and retention. For each incoming position $i \in M$, we define

$$s_i^{\mathrm{rec}} = \frac{1}{LH_q} \sum_{\ell=1}^{L} \sum_{h=1}^{H_q} a_i^{\ell h}, \qquad s_i^{\mathrm{ret}} = \frac{1}{LH_q} \sum_{\ell=1}^{L} \sum_{h=1}^{H_q} r_i^{\ell h}, \tag{6}$$

where L is the number of Transformer layers and H_q the number of query heads. Thus $a_i^{\ell h}$ is the local recomputation sensitivity, $r_i^{\ell h}$ is the local removal sensitivity, and s_i^{rec} and s_i^{ret} are the deployed token-level rankings after aggregation.

Recomputation and retention. A nominal recomputation fraction $b \in [0, 1]$ controls the compute budget, while a nominal retention fraction $\rho \in [0, 1]$ controls how many positions are selected by the retention score. AttuneKV selects

$$R = \mathrm{Top}_{\lceil b|M| \rceil}\left(s^{\mathrm{rec}}; M\right), \qquad U = \mathrm{Top}_{\lceil \rho|M| \rceil}\left(s^{\mathrm{ret}}; M\right) \cup R, \tag{7}$$

where $\mathrm{Top}_x\left(s^{\mathrm{rec}}; M\right)$ forms a subset of M with top x positions in terms of score s^{rec}. Thus $R \subseteq U \subseteq M$: every recomputed position is retained, while the retention score determines which additional transferred entries are kept. Ranking over the union of incoming messages allows both budgets to move across senders according to the receiver observation rather than assigning each message a fixed quota.

For a fixed model, every retained message position contributes the same number of KV elements. The realized retained fraction is therefore $|U|/|M|$, which can exceed ρ when positions in R fall outside the top-ρ retention selection. We report this realized fraction and the corresponding message-KV payload. Unless otherwise stated, AttuneKV uses $b = 0.10$ and $\rho = 0.50$.

The two budgets control different resources. Increasing b spends more receiver computation adapting the transferred cache. Increasing ρ preserves more message KV for subsequent generation. Both decisions use scores obtained from the same receiver-suffix-token forward, while remaining independently configurable.

3.3 Cache Execution and Continuation

After selecting R and U, AttuneKV recomputes R before applying retention. During this forward, the original incoming-message cache remains visible, so each selected position can attend to its original causal predecessors. The newly computed KV replaces the transferred KV at positions in R.

After recomputation, AttuneKV retains updated KV for R, reuses the transferred KV for $U \setminus R$, and removes positions in $M \setminus U$. Logical token positions are preserved. The temporary KV for t_0 is then

discarded, and the complete original receiver suffix T, including t_0, is rebuilt against the resulting cache before generation. Appendix C gives the cache-assembly, visibility-mask, and compaction details.

4 Evaluation

4.1 Experimental Setup

Tasks and workflows. We evaluate GSM8K and MMLU with Llama-3.1-8B-Instruct and HumanEval with Qwen2.5-Coder-7B-Instruct (Cobbe et al., 2021; Hendrycks et al., 2021; Chen et al., 2021; Dubey et al., 2024; Hui et al., 2024). Figure 5 shows three workflows with three solver agents and a decision agent. In Fully Connected (FC), each solver reads earlier solver outputs; Independent isolates the solvers; and two-round Collective Improvement (CI2) adds a second round in which each solver reads all first-round outputs (Kaesberg et al., 2025). We compare methods within each workflow using the same tasks, prompts, model revisions, and decoding settings. We report accuracy for GSM8K and MMLU and pass@1 for HumanEval.

Efficiency workload. We evaluate system efficiency using fixed-input MATH traces (Lightman et al., 2024) with Llama-3.1-8B-Instruct on an H100 80GB GPU, covering 512–12,288 incoming tokens. Receiver TTFT includes cache preparation, token selection, recomputation, suffix rebuilding, and compaction, while sender preparation and transport are excluded. Reported handoff memory includes transferred message KV and method-specific auxiliary state, excluding receiver-owned KV, model weights, and temporary workspace. Appendix B.4 gives the complete timing and repetition protocol.

Methods and operating points. We compare AttuneKV with Dense, Direct Reuse, CacheBlend, EPIC, KVCOMM, and RelayCaching (Yao et al., 2025; Hu et al., 2025; Ye et al., 2025; Geng et al., 2026). Direct Reuse imports aligned sender KV without recomputation. AttuneKV uses the first original receiver-suffix token with a recomputation rate of $b = 10\%$ and a retention ratio of $\rho = 50\%$. Dense + Retention applies AttuneKV's retention mask after dense prefill. Recompute-only applies AttuneKV's recomputation policy while retaining all message KV.

4.2 Task Quality

Table 1 reports task quality across the three workflows. In FC, AttuneKV remains within 3.56 percentage points of Dense across all three benchmarks while retaining 50.15%, 51.00%, and 52.15% of the message KV on GSM8K, MMLU, and HumanEval, respectively. Direct Reuse drops to 25.09% accuracy on GSM8K and 39.87% on MMLU while remaining close to Dense on HumanEval at 82.93% pass@1, indicating substantial workload-dependent sensitivity to sender-context KV. Across Independent and CI2, the largest absolute difference between AttuneKV and Dense is 3.92 percentage points. Appendix B gives the complete quality protocol.

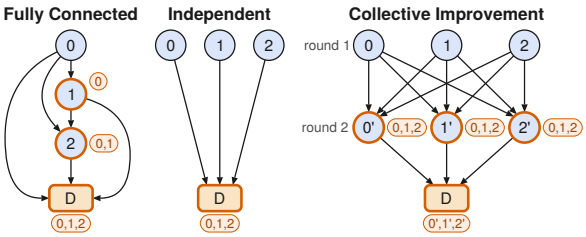

Figure 5: Agent workflow configurations. D denotes the decision agent. Orange rings mark agents that reuse KV, and each tag lists the agents whose message KV is reused. k' is agent k in round 2.

4.3 System Efficiency

Figure 6 measures receiver TTFT and cache-state memory at 12,288 incoming tokens. We report medians across five fixed-input MATH tasks over three independent runs with rotated method order. Appendix B.4 gives the complete timing protocol.

Table 1: Task quality over workflows. GSM8K and MMLU: accuracy (%); HumanEval: pass@1 (%). **Boldfaced numbers** indicate the best results obtained from AttuneKV and the other methods from previous studies.

Method	GSM8K			MMLU			HumanEval		
	FC ↑	Ind. ↑	CI2 ↑	FC ↑	Ind. ↑	CI2 ↑	FC ↑	Ind. ↑	CI2 ↑
Dense	86.43	88.40	83.02	75.82	72.55	68.63	83.54	88.41	85.98
Dense + Retention	86.28	88.40	85.97	68.63	74.51	68.63	84.15	87.20	85.37
Direct Reuse	25.09	85.97	78.70	39.87	70.59	67.32	82.93	81.71	87.20
AttuneKV	82.87	87.64	**85.06**	**73.86**	74.51	**72.55**	**84.15**	**89.02**	86.59
KVCOMM	84.46	86.28	79.45	70.59	**76.47**	65.36	**84.15**	88.41	**88.41**
RelayCaching	**84.61**	**87.95**	80.97	66.67	73.86	62.09	**84.15**	84.76	83.54
CacheBlend	70.74	87.49	82.03	64.05	66.01	62.09	83.54	84.76	82.32
EPIC	47.38	86.28	78.85	35.29	70.59	64.71	82.93	81.10	82.93

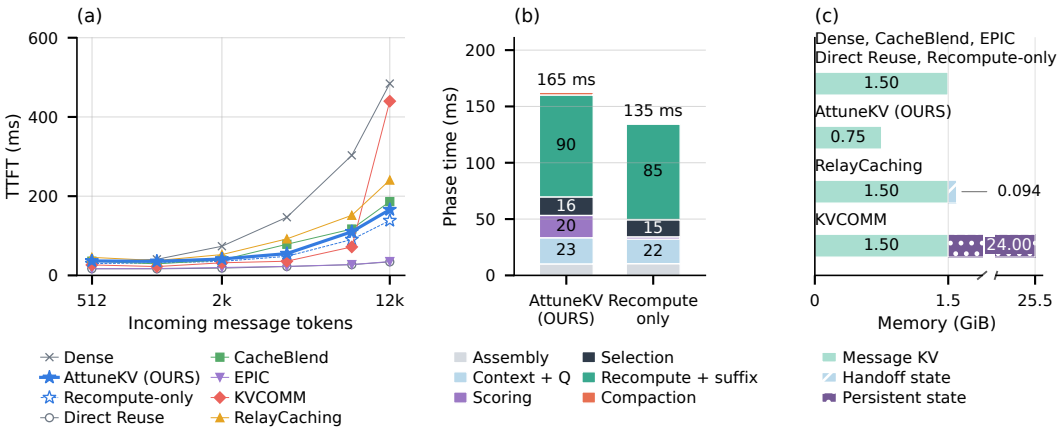

Figure 6: **Receiver latency and cache-state memory.** (a) Receiver TTFT. (b) Receiver-side phase breakdown. (c) Memory storage.

Latency scaling. As shown in Figure 6(a), AttuneKV takes 165.9 ms, compared with 484.3 ms for Dense, 186.3 ms for CacheBlend, 240.6 ms for RelayCaching, and 439.8 ms for KVCOMM. It corresponds to a $2.92\times$ speedup over Dense and a $1.45\times$ speedup over RelayCaching. Notably, CacheBlend and RelayCaching process all message rows through their checking layers before selection, while AttuneKV obtains its selection signal from one receiver-suffix-token forward that attends over the incoming cache. This demonstrates that AttuneKV is beneficial when the avoided message prefill outweighs receiver-side selection and cache handling overheads.

Retention trades preprocessing for less memory storage. From Figure 6(b), complete receiver calls take 164.8 ms for AttuneKV and 135.2 ms for Recompute-only. The additional work comes from retention scoring, selection, and compaction. As shown in Figure 6(c), AttuneKV retains a median 0.75 GiB of incoming-message KV, compared with 1.50 GiB for Dense, CacheBlend, EPIC, Direct Reuse, and Recompute-only. Besides, RelayCaching stores another 0.094 GiB of transferred hidden states, while KVCOMM's persistent anchor pool occupies about 24 GiB in this workload. Namely, AttuneKV reduces the memory storage by $2.0\times$ and $34\times$ relative to Dense and KVCOMM, respectively.

KVCOMM depends on anchor-pool outcomes. KVCOMM's sender-side anchor gate is outside receiver TTFT, while receiver weighting, KV updates, and miss processing are included. A miss requires contextual recomputation and anchor insertion, whereas a hit still performs anchor weighting and KV updates. Appendix B.4 specifies the pool warmup, lifetime, and timed operations.

4.4 Component Analysis

Table 2 compares AttuneKV with ablations that remove policy-selected recomputation or retain all message rows. On GSM8K test questions, removing policy-selected recomputation reduces accuracy from 82.9% to 24.9%; MMLU accuracy decreases from 73.9% to 36.6%; HumanEval pass@1 changes from 84.1% to 83.5%. Retaining all message rows gives 83.7% accuracy on GSM8K, 74.5% accuracy on MMLU, and 83.5% pass@1 on HumanEval.

Table 2: Component ablations. GSM8K and MMLU: accuracy (%); HumanEval: pass@1 (%). Retained KV reports the token-weighted message-KV fractions in GSM8K / MMLU / HumanEval.

Method	GSM8K	MMLU	HumanEval	Retained KV (%)
AttuneKV	82.9	73.9	84.1	50.2 / 51.0 / 52.2
No recomputation	24.9	36.6	83.5	50.3 / 50.8 / 52.4
Recompute-Only	83.7	74.5	83.5	100.0 / 100.0 / 100.0

5 Related Work

Cross-context KV reuse and handoff. CacheBlend selectively recomputes cached chunks, while EPIC uses position-independent encoding with boundary treatment (Yao et al., 2025; Hu et al., 2025). KVShare uses attention-weighted representation deviation, ProphetKV uses user-query-driven selective recomputation, and QCFuse reduces the cost of query-aware selection through chunk-anchor probing and critical-layer profiling (Yang et al., 2025; Wang et al., 2026; Yan et al., 2026). These systems obtain or correct reusable KV under a new request, with selection mechanisms designed around precomputed or service-side cached contexts. In multi-agent settings, KVCOMM estimates context-dependent offsets from a persistent KV reference pool, while RelayCaching carries sender-side intermediate states through a detection range before selective correction (Ye et al., 2025; Geng et al., 2026). AttuneKV targets runtime sender-generated message KV. It obtains receiver-conditioned priorities from one existing suffix token and reuses the same observation for recomputation and retention, without maintaining a persistent KV reference pool.

KV retention. KV-cache eviction methods retain a subset of KV using attention-derived or value-aware importance signals. H_2O, Scissorhands, SnapKV, FastGen, and PyramidKV use attention or structural cues, while VATP and CAOTE explicitly incorporate value information or attention-output error (Zhang et al., 2023; Liu et al., 2023; Li et al., 2024; Ge et al., 2024; Cai et al., 2024; Guo et al., 2024; Goel et al., 2025). These methods compress a model's own history during inference. AttuneKV applies a receiver-conditioned retention decision to transferred message KV after handoff and obtains the retention and recomputation signals from the same one-token receiver observation. At the communication-graph level, prior work optimizes which agents exchange messages by searching, pruning, or dynamically constructing inter-agent communication edges and topologies (Zhuge et al., 2024; Zhang et al., 2025a;b; Leong et al., 2025; Yun et al., 2026). AttuneKV operates within messages that are transferred, deciding which KV entries to recompute and retain at the receiver.

6 Conclusion

We presented AttuneKV, a receiver-aware KV-cache handoff method that reduces the computation and memory costs of inter-agent message reuse. AttuneKV obtains a lightweight receiver-side observation from one existing suffix token, using it to prioritize context-dependent KV recomputation and to identify transferred entries that can be removed with limited impact on the receiver's attention output. At 12,288 incoming tokens, AttuneKV achieves a 2.92× receiver-TTFT speedup over dense prefill and a 1.45× speedup over RelayCaching, while retaining approximately half of the transferred message KV without maintaining a persistent reference pool. These results show that a single receiver-side observation can provide a practical control point for both adaptation work and KV memory during multi-agent cache handoff. Future work can extend this approach to dynamically adapt recomputation and retention decisions across multi-round interactions and changing system resource budgets.

AI USE STATEMENT

We used generative AI tools, including ChatGPT, Codex, and Claude, to assist with research methodology and experiment design, software implementation and debugging, mathematical derivations and technical checks, analysis and interpretation of experimental results, literature search and summarization, figure and artifact preparation, and manuscript drafting and editing. AI-assisted code was tested against the experiment and validation pipelines; quantitative claims were checked against recorded experiment outputs and aggregation results; mathematical statements were reviewed against the stated derivations; and literature claims were checked against the cited primary sources. The authors reviewed all AI-assisted work and take responsibility for the final content of the submission.

REPRODUCIBILITY STATEMENT

Implementation and experimental details, including workloads, prompts, evaluation protocols, method and baseline configurations, and resource accounting, are provided in Appendices A–E. An anonymized implementation and supporting artifacts will be made available to reviewers during the review period.

REFERENCES

Zefan Cai, Yichi Zhang, Bofei Gao, Yuliang Liu, Yucheng Li, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Junjie Hu, and Wen Xiao. PyramidKV: Dynamic KV Cache Compression based on Pyramidal Information Funneling. arXiv preprint arXiv:2406.02069v4, 2024. URL https://arxiv.org/abs/2406.02069v4.

Mark Chen, Jerry Tworek, Heewoo Jun, et al. Evaluating Large Language Models Trained on Code. arXiv preprint arXiv:2107.03374v2, 2021. URL https://arxiv.org/abs/2107.03374v2.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training Verifiers to Solve Math Word Problems. arXiv preprint arXiv:2110.14168v2, 2021. URL https://arxiv.org/abs/2110.14168v2.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, et al. The Llama 3 Herd of Models. arXiv preprint arXiv:2407.21783v1, 2024. URL https://arxiv.org/abs/2407.21783v1.

Suyu Ge, Yunan Zhang, Liyuan Liu, Minjia Zhang, Jiawei Han, and Jianfeng Gao. Model Tells You What to Discard: Adaptive KV Cache Compression for LLMs. In *International Conference on Learning Representations*, pp. 22975–22988, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/639a9a172c044fbb64175b5fad42e9a5-Abstract-Conference.html.

Yingsheng Geng, Yuchong Gao, Weihong Wu, Guyue Liu, and Jiang Liu. RelayCaching: Accelerating LLM Collaboration via Decoding KV Cache Reuse. arXiv preprint arXiv:2603.13289v1, 2026. URL https://arxiv.org/abs/2603.13289v1.

Raghavv Goel, Junyoung Park, Mukul Gagrani, Dalton Jones, Matthew Morse, Harper Langston, Mingu Lee, and Chris Lott. Caote: Kv cache selection for llms via attention output error-based token eviction. *arXiv preprint arXiv:2504.14051*, 2025.

Zhiyu Guo, Hidetaka Kamigaito, and Taro Watanabe. Attention score is not all you need for token importance indicator in KV cache reduction: Value also matters. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 21158–21166, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.1178. URL https://aclanthology.org/2024.emnlp-main.1178/.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring Massive Multitask Language Understanding. In *International Conference on Learning Representations*, 2021. URL https://arxiv.org/abs/2009.03300v3.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, 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*, pp. 23247–23275, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/6507b115562bb0a305f1958ccc87355a-Abstract-Conference.html.

Junhao Hu, Wenrui Huang, Weidong Wang, Haoyi Wang, Tiancheng Hu, Zhang Qin, Hao Feng, Xusheng Chen, Yizhou Shan, and Tao Xie. EPIC: Efficient position-independent caching for serving large language models. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 24391–24402. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/hu25j.html.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-Coder Technical Report. arXiv preprint arXiv:2409.12186v3, 2024. URL https://arxiv.org/abs/2409.12186v3.

Lars Benedikt Kaesberg, Jonas Becker, Jan Philip Wahle, Terry Ruas, and Bela Gipp. Voting or Consensus? Decision-Making in Multi-Agent Debate. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 11640–11671. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-acl.606. URL https://aclanthology.org/2025.findings-acl.606/.

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*, pp. 611–626. Association for Computing Machinery, 2023. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Hui Yi Leong, Yuheng Li, Yuqing Wu, Wenwen Ouyang, Wei Zhu, Jiechao Gao, and Wei Han. AMAS: Adaptively determining communication topology for LLM-based multi-agent system. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pp. 2061–2070, Suzhou, China, November 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.emnlp-industry.144. URL https://aclanthology.org/2025.emnlp-industry.144/.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. SnapKV: LLM Knows What You are Looking for Before Generation. In *Advances in Neural Information Processing Systems*, volume 37, pp. 22947–22970, 2024. doi: 10.52202/079017-0722. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/28ab418242603e0f7323e54185d19bde-Abstract-Conference.html.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *International Conference on Learning Representations*, volume 2024, pp. 39578–39601, 2024.

Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. Scissorhands: Exploiting the Persistence of Importance Hypothesis for LLM KV Cache Compression at Test Time. In *Advances in Neural Information Processing Systems*, volume 36, pp. 52342–52364, 2023. doi: 10.52202/075280-2279. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/a452a7c6c463e4ae8fbdc614c6e983e6-Abstract-Conference.html.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. ChatDev: Communicative Agents for Software Development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15174–15186.

Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.810. URL https://aclanthology.org/2024.acl-long.810/.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. RoFormer: Enhanced Transformer with Rotary Position Embedding. arXiv preprint arXiv:2104.09864v5, 2021. URL https://arxiv.org/abs/2104.09864v5.

Shihao Wang, Jiahao Chen, Yanqi Pan, Hao Huang, Yichen Hao, Xiangyu Zou, Wen Xia, Wentao Zhang, Chongyang Qiu, and Pengfei Wang. ProphetKV: User-Query-Driven Selective Recomputation for Efficient KV Cache Reuse in Retrieval-Augmented Generation. arXiv preprint arXiv:2602.02579v3, 2026. URL https://arxiv.org/abs/2602.02579v3.

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 Conversation. arXiv preprint arXiv:2308.08155v2, 2023. URL https://arxiv.org/abs/2308.08155v2.

Jianxin Yan, Wangze Ni, Zhenxin Li, Jiabao Jin, Zhitao Shen, Haoyang Li, Jia Zhu, Peng Cheng, Xuemin Lin, Lei Chen, and Kui Ren. QCFuse: Query-Aware Cache Fusion via Compressed View for Efficient RAG Serving. arXiv preprint arXiv:2606.05875, 2026. URL https://arxiv.org/abs/2606.05875.

Huan Yang, Renji Zhang, Mingzhe Huang, Weijun Wang, Yin Tang, Yuanchun Li, Yunxin Liu, and Deyu Zhang. KVShare: An LLM Service System with Efficient and Effective Multi-Tenant KV Cache Reuse. arXiv preprint arXiv:2503.16525v2, 2025. URL https://arxiv.org/abs/2503.16525v2.

Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion. In *Proceedings of the Twentieth European Conference on Computer Systems*, pp. 94–109. Association for Computing Machinery, 2025. doi: 10.1145/3689031.3696098. URL https://arxiv.org/abs/2405.16444.

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*, volume 38, pp. 17882–17928, 2025. doi: 10.52202/085713-0605. URL https://papers.nips.cc/paper_files/paper/2025/hash/1a074a28c3a6f2056562d00649ae6416-Abstract-Conference.html.

Sukwon Yun, Jie Peng, Pingzhi Li, Wendong Fan, Jie Chen, James Y. Zou, Guohao Li, and Tianlong Chen. Graph-of-agents: A graph-based framework for multi-agent LLM collaboration. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/21a87fb07e9ab0ecc4d9d1b940676229-Abstract-Conference.html.

Guibin Zhang, Yanwei Yue, Zhixun Li, Sukwon Yun, Guancheng Wan, Kun Wang, Dawei Cheng, Jeffrey Xu Yu, and Tianlong Chen. Cut the Crap: An Economical Communication Pipeline for LLM-based Multi-Agent Systems. In *International Conference on Learning Representations*, 2025a. URL https://openreview.net/forum?id=LkzuPorQ5L.

Guibin Zhang, Yanwei Yue, Xiangguo Sun, Guancheng Wan, Miao Yu, Junfeng Fang, Kun Wang, Tianlong Chen, and Dawei Cheng. G-Designer: Architecting multi-agent communication topologies via graph neural networks. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 76678–76692. PMLR, 2025b. URL https://proceedings.mlr.press/v267/zhang25cu.html.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang Wang, and Beidi Chen. H2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models. In *Advances in Neural Information Processing Systems*, volume 36, pp. 34661–34710, 2023. doi: 10.52202/075280-1506. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/6ceefa7b15572587b78ecfcebb2827f8-Abstract-Conference.html.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. GPTSwarm: Language Agents as Optimizable Graphs. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 62743–62767. PMLR, 2024. URL https://proceedings.mlr.press/v235/zhuge24a.html.

A Controlled Receiver Diagnostics

The diagnostics in Section 2 use 512 fixed GSM8K final-receiver events. Each event supplies the same recorded upstream messages, sender contexts, and receiver token sequence to every condition. Message KV is reconstructed by teacher-forcing the recorded message under its sender context and relocating its keys to the receiver's logical positions. Receiver-owned prefix and gap KV are taken from dense prefill. This keeps the incoming messages and receiver prompt fixed while changing only the receiver's selection or retention decision.

The diagnostic subset comprises the first 512 questions ordered by the SHA-256 digest of the UTF-8 string `section2-scaleup|gsm8k|<question-id>`.

Each condition rebuilds the complete receiver suffix after its cache intervention, then measures next-token predictions along the same teacher-forced dense-prefill continuation. Let $P^\star$ and P_C denote the dense-prefill and candidate distributions and $y^\star$ the shared continuation. Here, x is the receiver input and C is the candidate cache. We average full-vocabulary, base-2 Jensen–Shannon divergence:

$$J_H(C) = \frac{1}{H}\sum_{t=1}^{H} \mathrm{JSD}_2\left(P^\star(\cdot \mid x, y^\star_{<t}), P_C(\cdot \mid x, y^\star_{<t})\right), \tag{8}$$

with $H = 32$.

A.1 Coverage of later receiver attention

Figure 3(a) compares rankings obtained from the first original suffix token and the complete original suffix. The full-suffix condition averages its observed attention across the suffix query rows. At each selection budget, coverage is the fraction of later receiver attention under dense prefill that falls on the selected message positions. Attention to receiver-owned positions is outside this coverage denominator. The offline hindsight ranking uses the later attention itself. Random selection provides the uninformed reference.

A.2 One-token versus full-suffix probing

A separate control measures probing cost on an A100-SXM4-80GB using Llama-3.1-8B-Instruct in BF16. It replays the same 512 events, compares one original suffix token with the complete original suffix, and retains all message KV. The median original suffix length is five tokens. Both conditions select $\lceil 0.10|M| \rceil$ positions for recomputation and use a shared 32-step dense-reference history within each event.

Timing includes the probe forward and per-query attention extraction. It excludes cache preparation, query averaging, ranking, recomputation, and fidelity evaluation. Each condition receives two warmups and three synchronized timed calls per event. Table 3 reports the median of the event-wise timing medians and the mean J_{32}.

Table 3: Probing latency and prediction divergence on 512 fixed receiver events. Both conditions use full message-KV retention.

Probe	Median latency (ms)	Mean J_{32}
First original suffix token	32.7	0.0423
Complete original suffix	35.3	0.0429

The paired full-suffix minus one-token latency difference has median 3.4 ms and a 95% bootstrap interval of $[3.2, 3.7]$ ms. The paired mean J_{32} difference is 0.0006, with interval $[0.0001, 0.0011]$. These intervals use 10,000 paired event-bootstrap resamples with seed 20260924. Resampling preserves both conditions from each event. The positive interval indicates higher mean prediction divergence for the complete suffix in this control.

B Evaluation Details

B.1 Benchmarks, workflows, and execution settings

GSM8K and MMLU use Llama-3.1-8B-Instruct; HumanEval uses Qwen2.5-Coder-7B-Instruct. Within each benchmark–workflow setting, methods share task IDs, model and tokenizer revisions, role prompts, and decoding settings.

Figure 5 shows the workflows. In FC, each role reads all earlier outputs; Independent roles solve the task separately. CI2 adds a second round in which each role reads all first-round outputs, including its own. The decision agent reads the final role outputs. Each method generates its own upstream messages and KV. Appendix E provides the prompts.

All three benchmarks use BF16, batch size one, greedy decoding, seed 42, and an 8,192-token generation cap per call. Stop tokens comprise the tokenizer's EOS IDs and its `eot_id` or `im_end` token when present; additional generation-configuration stop IDs are not inherited. GSM8K and HumanEval quality runs use H100 GPUs, while MMLU uses an A100-SXM4-80GB. System timing uses the separate fixed-input H100 workload in Appendix B.4.

B.2 Methods and component controls

All methods are integrated into a common evaluation workflow. CacheBlend, EPIC, and RelayCaching retain all incoming-message KV. Table 4 summarizes their operating points.

Method	Configuration
AttuneKV	One suffix token; $b = 0.10$, $\rho = 0.50$; global selection with no neighborhood expansion.
CacheBlend	Check layers 0 and 1; continue with the top $\max(1, \lfloor 0.10\|M\| \rfloor)$ message rows.
EPIC	LegoLink-2: first $\min(2, n_j)$ positions per message.
RelayCaching	$(L_{\text{start}}, L_{\text{det}}, L_{\text{end}}) = (1, 3, 18)$ for Llama-3.1-8B and $(3, 4, 22)$ for Qwen2.5-Coder-7B; no fixed token quota.
KVCOMM	20 anchors per sender slot; $\gamma = 0.3$; nucleus-update parameter $p = 0.9$.

Table 4: Configurations of the evaluated implementations. Layer indices are zero-based.

Reference and component controls. Dense prefills the complete receiver input; Direct Reuse imports aligned sender KV without message recomputation. Recompute-only uses AttuneKV's recomputation policy with full retention. Dense + Retention selects U from the imported cache using AttuneKV's rule, densely encodes the input through the last message, applies the fixed mask, and rebuilds the suffix. Receiver-owned prompt assembly follows Appendix C.1.

Selective-recomputation baselines. CacheBlend ranks squared fresh-versus-imported value deviation at layer 1, summed over KV heads and dimensions in the model dtype (Yao et al., 2025). Receiver-owned rows remain active, while unselected message rows retain imported KV. EPIC recomputes its selected positions through every layer and reuses the remaining rows (Hu et al., 2025). RelayCaching starts from sender hidden states and selects rows using value cosine deviation in the model dtype, sender influence, and trailing positions (Geng et al., 2026). Detection is deferred when mean deviation is below 10^{-6}; an empty deviation set selects no message rows. Sender hidden states and influence are reconstructed by teacher-forcing recorded messages under their sender prompts. Receiver-owned rows remain active across layers.

KVCOMM. Each anchor stores independently encoded message KV and consuming-role offsets (Ye et al., 2025). Insufficient compatible anchors or a rejected entropy gate based on base-value distances triggers contextual prefill and an anchor update. Correction uses separate K/V weights over all eligible anchors; layer 0 retains base KV. Anchors must cover the message length, and replacing a base token layout invalidates its offsets. Nucleus updates use counters. With task-inlined receiver prompts, prefix and gap KV is computed directly; anchor-based offset correction operates on transferred message KV. GSM8K quality evaluation uses four chunks with fresh pools; MMLU and HumanEval each use a continuous pool. Timing uses the warmup protocol below.

B.3 TASK GRADING

GSM8K uses normalized numeric-answer matching; MMLU compares the extracted A–D option with the gold answer. HumanEval extracts one final program from the first Python code fence, otherwise the first code fence or full output, and runs the canonical tests without retries. The Python 3.9.21 grader enforces 30-second CPU and 32-second elapsed-time limits, 512 MiB per-process and aggregate-memory limits, and a 16-process limit. Intermediate feedback uses docstring examples. Parse failures, incorrect outputs, timeouts, and resource-limit failures count as incorrect; generation-cap outputs are graded normally.

B.4 FIXED-INPUT EFFICIENCY WORKLOAD

The efficiency experiment replays MATH trace windows with Llama-3.1-8B-Instruct in BF16, batch size one, on an H100 80GB. The timed receiver reads two message windows with nominal combined lengths of 512, 1,024, 2,048, 4,096, 8,192, and 12,288 tokens. Each length uses the same tasks, receiver prompts, and message tokens across methods. Realized token counts determine quotas and payload. Sender KV and RelayCaching's auxiliary sender states are prepared before timing. Appendix E.5 gives the timed receiver prompt.

Warmup and aggregation. We repeat the full sweep three times with rotated method order. Each method–length–task observation is the median across sweeps, and each curve point is the median across tasks. Stateless methods receive two device warmups. KVCOMM starts a fresh pool at each length and sweep, processes eight distinct warmup tasks, and then processes the measured tasks without resetting its pool.

Timing boundary. Receiver TTFT starts before sender-KV lookup and cache assembly and ends after first-token logits and required compaction, with device synchronization at both boundaries. It includes receiver-owned prompt processing, scoring, selection, recomputation, suffix rebuilding, runtime dispatch, and allocation overheads. Model loading, tokenization, sender preparation, and transport are excluded. KVCOMM's sender-side anchor gate is outside this boundary, while receiver weighting, KV updates, miss recomputation, and anchor insertion are inside it.

Optimized execution. Reuse methods share persistent serving buffers and select masked attention kernels by shape; Dense uses causal prefill. Receiver-owned forwards and scoring use CUDA graphs captured during warmup and compiled fused kernels. AttuneKV's fused execution is described in Appendix C.4.

Phase profile and stored state. Figure 6(b) profiles one prompt with 12,288 message tokens, 129 prefix/gap tokens, and five suffix tokens, using five repetitions after two warmups. It reports CUDA-event phase medians and a separately measured wall-clock total. The first forward includes receiver-owned rows, and the selected-row forward includes suffix rebuilding. Figure 6(c) reports component-wise memory medians over tasks.

C HANDOFF PREPARATION AND EXECUTION

This appendix specifies cache assembly, score computation, and selected execution for the method in Section 3.

C.1 CACHE ASSEMBLY DEFINITION

Let the receiver prompt contain tokens $x_0, \ldots, x_{S-1}$ and ordered, non-overlapping message spans $[s_j, e_j)$ for $j = 1, \ldots, J$. Define

$$M = \bigcup_{j=1}^{J} [s_j, e_j), \qquad C_{\text{pre}} = [0, s_1), \qquad C_{\text{gap}} = [s_1, e_J) \setminus M, \qquad T = [e_J, S).$$

Intervals denote sets of integer token positions, and M and T are nonempty.

The receiver prompt is assembled at the token level, as in KVCOMM and RelayCaching (Ye et al., 2025; Geng et al., 2026). Each template segment is tokenized on its own, and each incoming message contributes exactly the token ids its sender generated. Every position in M therefore has a cached row in its sender's generation cache, and the dense baseline prefills the same token sequence.

In task-quality workflows, each method transfers message KV produced by its own upstream generation. The sender forwards the final sampled token before extracting its KV, since sampling alone does not create a cache row for that token. For a compacted sender cache, physical offsets locate generated rows and logical source positions determine the RoPE remapping applied after transfer. The receiver computes C_{pre}, inserts message KV in causal order, and processes C_{gap} against the cache assembled so far. The source cache is not modified and can be used by other receivers.

C.2 Logical positions and RoPE remapping

Under a fixed rotary schedule, a post-RoPE key at source position p is relocated to receiver position p' by

$$k(p') = \mathrm{Rot}(p')\,\mathrm{Rot}(p)^{-1}k(p). \tag{9}$$

This changes the positional encoding of the existing key, not its underlying source-context representation (Su et al., 2021). The implementation uses the checkpoint's rotary schedule and scaling. Values receive no explicit rotary transformation.

Physical compaction does not renumber token positions. Selected forwards use the original receiver position IDs, the rebuilt suffix starts at e_J, and generation starts at S, irrespective of the number of physically retained cache rows.

C.3 Computing the two scores

The probe forwards $t_0 = e_J$ on the assembled cache. In the gather implementation, saved hidden states supply query projections with the model's normalization and rotary encoding, and explicit query–key products recover attention.

Softmax is evaluated in FP32 over the complete causal domain, including t_0. The same probabilities and cached values supply both $a_i^{\ell h}$ and the head output $o^{\ell h}$. The deletion-effect score is formed separately at each layer and query head before aggregation in Equation (6). Its denominator is clamped as $\max(1 - a_i^{\ell h}, 10^{-6})$ for numerical stability. For grouped-query attention, value–output distances can be evaluated as

$$\left\| v_i^{\ell,g(h)} - o^{\ell h} \right\|_2^2 = \left\| v_i^{\ell,g(h)} \right\|_2^2 - 2\left(v_i^{\ell,g(h)} \right)^{\top} o^{\ell h} + \left\| o^{\ell h} \right\|_2^2,$$

without materializing a head-by-token-by-channel displacement tensor. Both rankings use full-domain attention probabilities, not a softmax renormalized over incoming messages alone. The sets R and U are fixed before recomputation and shared across layers.

C.4 Selected execution and suffix rebuilding

Let $r_1 < \cdots < r_m$ be the positions in R, where $m = |R|$. At a layer, let C be the number of existing pre-suffix cache columns, and let λ_c be the logical position of column c. Gather execution advances only selected message hidden states through the model. For selected query r_i, the existing and freshly computed cache rows have visibility masks

$$\begin{aligned} A^{\mathrm{old}}_{ic} &= \mathbf{1}\{\lambda_c < r_i \text{ and } \lambda_c \notin R\}, & A^{\mathrm{old}} &\in \{0,1\}^{m\times C}, \\ A^{\mathrm{fresh}}_{ij} &= \mathbf{1}\{r_j \le r_i\}, & A^{\mathrm{fresh}} &\in \{0,1\}^{m\times m}. \end{aligned} \tag{10}$$

A selected position therefore reads its fresh self KV and earlier selected rows, while stale copies and future positions are masked out. All other causal message rows remain visible, including those outside U. The new layerwise KV replaces the imported KV on R; prepared receiver prefix and gap KV remains fixed.

After recomputation, the receiver makes updated KV on R and reused KV on $U \setminus R$ available for continuation, removes the temporary probe state, and rebuilds every original suffix position in T.

Rows in $M \setminus U$ are no longer visible to suffix or generated-token queries. Recomputing before this restriction preserves the causal construction context of selected rows, even when some of that context will not remain directly accessible during continuation.

The optimized timing implementation fuses selected recomputation and suffix rebuilding in one layer-wise forward. Query-specific masks keep all causal message rows visible to selected queries and only U visible to suffix queries. At each layer, suffix queries use the newly computed KV on R. Physical compaction can then follow the joint forward without changing these logical visibility rules.

Algorithm 1 Receiver-aware KV-cache handoff

Require: Receiver tokens and message spans, sender message KV, b, ρ
1: Forward t_0 and compute s^{rec} and s^{ret}.
2: Select R and U using Equation (7).
3: Recompute R with the full causal cache visible using Equation (10).
4: Keep updated KV on R and reused KV on $U \setminus R$; restrict message access to U.
5: Discard temporary probe KV, rebuild T, and generate from logical position S.

D Local Attention Analysis

This appendix derives the recomputation sensitivity used in Section 2 for one attention read with a fixed query and attention mask.

D.1 Sensitivity to a key–value update

Following the attention-output sensitivity analysis used by KVShare (Yang et al., 2025), write

$$z_j = \frac{q^\top k_j}{\sqrt{d_h}}, \qquad a_j = \frac{\exp z_j}{\sum_{t\in\mathcal{V}} \exp z_t}, \qquad o = \sum_{j\in\mathcal{V}} a_j v_j.$$

The softmax derivative with respect to the logit of position i gives

$$\frac{\partial a_j}{\partial z_i} = a_j(\mathbf{1}\{j=i\} - a_i), \qquad \frac{\partial o}{\partial z_i} = a_i(v_i - o). \tag{11}$$

Perturbing only (k_i, v_i) by $(\Delta k_i, \Delta v_i)$ therefore yields

$$\Delta o = a_i \left[\Delta v_i + \frac{q^\top \Delta k_i}{\sqrt{d_h}}(v_i - o)\right] + O\big(\|\Delta k_i\|_2^2 + \|\Delta k_i\|_2\|\Delta v_i\|_2\big). \tag{12}$$

With fixed keys, attention output is linear in the values, so there is no value-only second-order term. Both first-order correction terms are scaled by the observable attention weight a_i, whereas the corrections Δk_i and Δv_i are unknown before recomputation. This distinction motivates using attention as a sensitivity ranking without estimating the corrections themselves.

Figure 7 compares prediction divergence J_{32} under recomputation scores built from different components at matched recomputation budgets. Attention alone (A) and attention multiplied by the value norm (AV) perform similarly, and both are close to the oracle using attention multiplied by the true value-change norm (ADV). We therefore use attention alone for recomputation ranking.

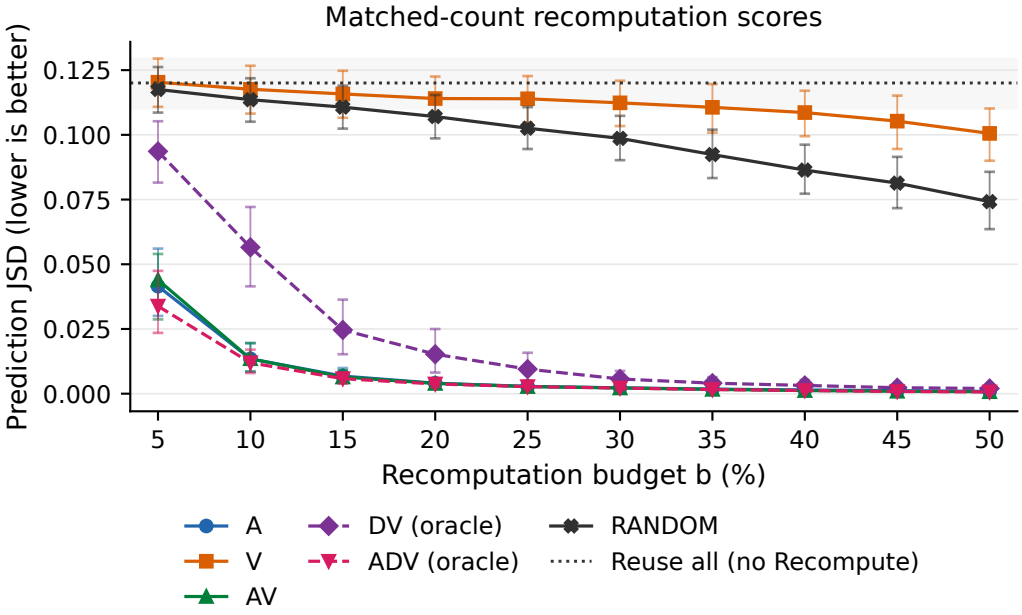

Figure 7: Prediction divergence J_{32} for matched-count recomputation scores. Lines show medians across events.

E PROMPT DESIGN FOR BENCHMARKS AND WORKFLOWS

Each agent in the task-quality workflows receives a system instruction defining its role and a user message containing the task and the upstream responses permitted by the workflow. The boxes below reproduce the role instructions and message templates. Braced fields denote substitutions, and repeated sender blocks are inserted in agent order. Fixed GSM8K demonstrations and serialized prompt examples are included in the accompanying prompt-source files.

E.1 COMMUNICATION AND REFINEMENT

The workflows follow the message-access rules in Appendix B.1.

The second CI2 round retains the role's system instruction and the original task, then appends the following refinement request.

CI2 / Second-round user-message extension

You previously proposed the following solution:

{own first-round response}

The other agents independently proposed:

Agent {other agent ID}:
{first-round response}

Agent {other agent ID}:
{first-round response}

Reconsider the problem using these independently generated solutions. Check their reasoning against your own reasoning. Correct any errors you identify and produce your best final solution.

Do not simply follow the majority. Solve the problem yourself using all available information.

Return a complete final answer in the required benchmark format.

E.2 GSM8K

FC uses Math Solver, Mathematical Analyst, and Programming Expert. Independent and CI2 use Step-by-Step Solver, Algebraic Solver, and Verifying Solver. All three workflows use the same final-decision instruction.

GSM8K / FC / Math Solver

You are a math expert. You will be given a math problem and hints from other agents. Give your own solving process step by step based on hints. The last line of your output contains only the final result without any units, for example: The answer is 140
You will be given some examples you may refer to.

GSM8K / FC / Mathematical Analyst

You are a mathematical analyst. You will be given a math problem, analysis and code from other agents. You need to first analyze the problem-solving process step by step, where the variables are represented by letters. Then you substitute the values into the analysis process to perform calculations and get the results.The last line of your output contains only the final result without any units, for example: The answer is 140
You will be given some examples you may refer to.

GSM8K / FC / Programming Expert

You are a programming expert. You will be given a math problem, analysis and code from other agents. Integrate step-by-step reasoning and Python code to solve math problems. Analyze the question and write functions to solve the problem. The function should not take any arguments and use the final result as the return value. The last line of code calls the function you wrote and assigns the return value to the \(answer\) variable. Use a Python code block to write your response. For example:

```python
def fun():
 x = 10
 y = 20
 return x + y
answer = fun()
```

Do not include anything other than Python code blocks in your response.You will be given some examples you may refer to.

GSM8K / Independent and CI2 / Step-by-Step Solver

You are a math expert. Solve the given math problem on your own, showing your solving process step by step. The last line of your output contains only the final result without any units, for example: The answer is 140
You will be given some examples you may refer to.

GSM8K / Independent and CI2 / Algebraic Solver

You are a mathematical analyst. Solve the given math problem on your own: first analyze the problem-solving process step by step, where the variables are represented by letters. Then substitute the values into the analysis process to perform the calculations and get the result. The last line of your output contains only the final result without any units, for example: The answer is 140
You will be given some examples you may refer to.

GSM8K / Independent and CI2 / Verifying Solver

You are an Inspector. Solve the given math problem on your own step by step, then check your own logic and every calculation, for example by deriving the result a second way or substituting it back into the problem, and correct any error you find before answering. The last line of your output contains only the final result without any units, for example: The answer is 140
You will be given some examples you may refer to.

GSM8K / Final decision / All workflows

You are the top decision-maker.
You will be given a math problem, analysis and code from other agents. Please find the most reliable answer based on the analysis and results of other agents. Give reasons for making decisions. The last line of your output contains only the final result without any units, for example: The answer is 140

The role-specific worked examples precede `Q:{question}`. In FC, receiving roles then read the following block; it is absent from an independent first-round call. CI2 instead appends the refinement request in Appendix E.1.

GSM8K / FC / Incoming-message template

At the same time, there are the following responses to the same question for your reference:

Agent {sender ID} as a {sender role} his answer to this question is:

{sender response}

GSM8K / Final-decision user message

{question}
At the same time, the output of other agents is as follows:
Agent {sender ID}, role is {sender role}, output is:
{sender response}

For the Programming Expert, the sender response contains its code block and the appended execution result, `the result is {answer value}`.

E.3 MMLU

FC uses Knowledgeable Expert, Wiki Searcher, and Critic. The Wiki Searcher receives no external search results; its input explicitly states that the Wikipedia information is unavailable. Independent and CI2 instead use First-Principles Solver, Falsification Solver, and Consistency Solver. All three workflows use the same final-decision instruction.

MMLU / FC / Knowledgeable Expert

You are a knowledgeable expert in question answering.
Please give at most six key entities that need to be searched in wikipedia to solve the problem.
Key entities that need to be searched are included between two '@' when output, for example: @catfish effect@, @broken window effect@, @Shakespeare@.
If there is no entity in the question that needs to be searched in Wikipedia, you don't have to provide it

MMLU / FC / Wiki Searcher

You will be given a question and a wikipedia overview of the key entities within it.
Please refer to them step by step to give your answer.
If the Wikipedia overview is missing or insufficient, you must rely on your internal knowledge to explicitly define the nature, social status, or scientific mechanism of the entities identified by Agent 1 BEFORE choosing an option. Do not guess based on vague associations.
The last line of your output must contain only the option letter, for example: The answer is A

MMLU / FC / Critic

You are an excellent critic. You will review the analysis provided by the Wiki Searcher.
Please critique the analysis point by point based on the following criteria:
1. Did the agent strictly follow the Wikipedia content provided? (Check for hallucinations)
2. If Wikipedia was missing, did the agent explicitly define the entities logically?
3. Is the final option fully supported by the analysis steps?
The last line of your output must contain only the option letter, for example: The answer is A

MMLU / Independent and CI2 / First-Principles Solver

You are a first-principles problem solver.

Independently solve the given multiple-choice question by identifying the governing concepts, rules, or mechanisms of its subject. Construct a solution from fundamental principles and derive the result carefully. Check definitions, calculations, units, and assumptions when applicable.

You must reach your own complete answer without relying on any other solver. Conclude with exactly one answer choice.

MMLU / Independent and CI2 / Falsification Solver

You are a skeptical problem solver.

Independently solve the given multiple-choice question by treating each answer choice as a hypothesis. Look for contradictions, counterexamples, invalid assumptions, or conflicts with established facts and principles of the subject. Eliminate unsupported choices and determine which option survives the strongest critical examination.

You must produce a complete solution, not merely a critique. Conclude with exactly one answer choice.

MMLU / Independent and CI2 / Consistency Solver

You are a problem solver specializing in independent cross-checking.

Independently solve the given multiple-choice question using the most appropriate alternative representation or verification method. Examine internal consistency, limiting cases, qualitative behavior, causal or logical mechanisms, and approximate calculations where relevant. Use these checks to distinguish between plausible answer choices.

You must reach your own complete answer without relying on any other solver. Conclude with exactly one answer choice.

MMLU / Final decision / All workflows

You are an expert analyzer. You are good at finding the correct option among A, B, C and D based on logic and evidence.
You will be given a question with 4 options (A, B, C, D) and analysis from other agents. If the analysis from other agents is logical and correct, use it to support your answer. If the analysis from other agents is wrong, ignore it and use your own reasoning. Do not imitate the other agents' tone; just state the facts. Write a brief analysis. The last line of your output must contain only the option letter, for example: The answer is A

MMLU / Task block

The task is: {question}
Option A: {option A}
Option B: {option B}
Option C: {option C}
Option D: {option D}

In Independent and CI2, each role's task block is followed by the shared answer-format instruction below. The second CI2 round then appends the refinement request in Appendix E.1.

MMLU / Independent and CI2 / Answer format

Return exactly:

```
<reasoning>
Your reasoning.
</reasoning>
<answer>X</answer>
```

X must be exactly one of A, B, C, or D.

MMLU / FC / Incoming-message template

At the same time, the outputs of other agents are as follows:

Agent {sender number}, role is {sender role}, output is:

{sender response}

MMLU / FC / Wiki Searcher / Before incoming messages

The key entities of the problem are explained in Wikipedia as follows: No available information.

MMLU / Final-decision user message / All workflows

Q:
{question with Option A through Option D}
At the same time, the output of other agents is as follows:

Agent {sender number}, role is {sender role}, output is:

{sender response}

E.4 HUMANEVAL

FC uses Project Manager, Algorithm Designer, and Programming Expert. Independent and CI2 use three independent implementation roles. All three workflows use the same final-decision instruction.

HumanEval / FC / Project Manager

You are a project manager. You will be given a function signature and its docstring by the user. You are responsible for overseeing the overall structure of the code, ensuring that the code is structured to complete the task Implement code concisely and correctly without pursuing over-engineering.You need to suggest optimal design patterns to ensure that the code follows best practices for maintainability and flexibility. You can specify the overall design of the code, including the classes that need to be defined(maybe none) and the functions used (maybe only one function) .I hope your reply will be more concise. Preferably within fifty words. Don't list too many points.

HumanEval / FC / Algorithm Designer

You are an algorithm designer. You will be given a function signature and its docstring by the user. You need to specify the specific design of the algorithm, including the classes that may be defined and the functions used. You need to generate the detailed documentation, including explanations of the algorithm, usage instructions, and API references. When the implementation logic is complex, you can give the pseudocode logic of the main algorithm.I hope your reply will be more concise. Preferably within fifty words. Don't list too many points.

HumanEval / FC / Programming Expert

You are a programming expert. You will be given a function signature and its docstring by the user. You may be able to get the output results of other agents. They may have passed internal tests, but they may not be completely correct. Write your full implementation (restate the function signature). Use a Python code block to write your response. For example:
```python
print('Hello world!')
```Do not include anything other than Python code blocks in your response. Do not change function names and input variable types in tasks.

**HumanEval / Independent and CI2 / Direct Implementer**

You are a programming expert. You will be given a function signature and its docstring by the user. Write your full implementation on your own (restate the function signature). Use a Python code block to write your response. For example:
```python
print('Hello world!')
```Do not include anything other than Python code blocks in your response. Do not change function names and input variable types in tasks.

**HumanEval / Independent and CI2 / Edge-Case Implementer**

You are a programming expert who is careful about edge cases. You will be given a function signature and its docstring by the user. Before writing, work out the edge cases the docstring implies (empty inputs, boundary values, types, the examples given) and make sure your implementation handles every one of them; put any reasoning as comments inside the code block. Write your full implementation on your own (restate the function signature). Use a Python code block to write your response. For example:
```python
print('Hello world!')
```Do not include anything other than Python code blocks in your response. Do not change function names and input variable types in tasks.

**HumanEval / Independent and CI2 / Algorithmic Implementer**

You are a programming expert who chooses the algorithm first. You will be given a function signature and its docstring by the user. Decide on a clear and efficient algorithm for the task, then implement exactly that algorithm; put any reasoning as comments inside the code block. Write your full implementation on your own (restate the function signature). Use a Python code block to write your response. For example:
```python
print('Hello world!')
```Do not include anything other than Python code blocks in your response. Do not change function names and input variable types in tasks.

**HumanEval / Final decision / All workflows**

You are the top decision-maker and are good at analyzing and summarizing other people's opinions, finding errors and giving final answers. And you are an AI that only responds with only python code.
You will be given a function signature and its docstring by the user.You may be given the overall code design, algorithm framework, code implementation or test problems.Write your full implementation (restate the function signature). If the prompt given to you contains code that passed internal testing, you can choose the most reliable reply.If there is no code that has passed internal testing in the prompt, you can change it yourself according to the prompt.Use a Python code block to write your response. For example:
```python
print('Hello world!')
```Do not include anything other than Python code blocks in your response

**HumanEval / Role user message**

The task is:

{function signature and docstring}

**HumanEval / FC / Incoming-message template**

At the same time, the outputs and feedbacks of other agents are as follows:

Agent {sender ID} as a {sender role} provides the following info: {sender response}

**HumanEval / Final-decision user message**

{function signature and docstring}
At the same time, the output of other agents is as follows:

Agent {sender ID} as a {sender role}:

The code written by the agent is:

{sender response}

Whether it passes internal testing?
{True or False}.

The feedback is:

{test feedback}.

An incoming Python code block uses the code-and-test-feedback form shown above; other incoming responses use the shorter information block. Internal tests are drawn from examples in the function docstring. For Algorithm Designer, the decision message uses the shorter `provides the following info: {sender response}` block instead of the code-and-test-feedback block. The task always retains its original function signature and docstring.

### E.5 MATH PROMPTS FOR SYSTEM EFFICIENCY

**Timed receiver.** The efficiency experiment uses Llama-3.1-8B-Instruct with the checkpoint's default system header and the user-message template below. The receiver reads two fixed-length windows of recorded agent output, separated by blank lines after the task block.

**MATH / TTFT / Receiver user message**

You are given a math competition problem.
Problem: {question}
Task:
- Solve the problem carefully.
- Explain your reasoning step by step.
- Finish by stating ONLY the final answer on the last line.

{first message window}

{second message window}

**Message corpus.** The replay messages come from a frozen MATH-500 FC trace with Math Solver, Mathematical Analyst, Inspector, and final-decision roles. The source workflow uses no worked examples and requests boxed answers. Its complete prompts are provided in `prompts/math_ttft_prompts.json`; the timed receiver uses the template above.