FORECACHE: EFFICIENT LLM-BASED MULTI-AGENT SYSTEMS WITH INTENT-CONDITIONED PRUNING AND AMORTIZED KV REUSE

Anonymous authors
Paper under double-blind review

ABSTRACT

Multi-agent systems based on large language models (LLMs) address complex tasks such as code generation by exchanging and refining intermediate outputs, but repeatedly processing accumulated interaction histories leads to high inference latency. Existing approaches reduce this overhead through history compression or key-value (KV) cache reuse. However, compression can remove information needed by the next agent, while direct KV reuse can reduce downstream answer accuracy when cached states no longer match that agent's context. To address these limitations, we propose ForeCache, a training-free framework for efficient multi-agent inference. Specifically, consumer-conditioned saliency (CCS) uses each downstream agent's instructions to estimate the task relevance of shared history and guide pruning. We also introduce divergence-guided recomputation (DGR), which reuses upstream KV states and stops further recomputation for tokens whose cached and updated states are sufficiently close, reducing repeated history encoding. Experiments on well-known benchmarks show that ForeCache maintains task performance comparable to full prefilling. In the six-agent setting, it achieves up to a $6.6\times$ speedup in the final agent's time to first token (TTFT) over full prefilling.

1 INTRODUCTION

LLM-based multi-agent systems pass intermediate outputs among agents to solve complex tasks, but repeated prefilling of shared history slows inference. Recent systems organize agents in layers or communication graphs to exchange and refine intermediate outputs (Wang et al., 2025a; Qian et al., 2025; Yun et al., 2026). When these agents use the same LLM, each agent prepends its own system prompt to the history received from other agents. Because the prompts differ, the same history is processed under a different prefix at each agent, making its previously computed key-value (KV) states inconsistent with the new context. Re-prefilling this shared history repeats computation and increasingly delays responses as the interaction history grows (Ye et al., 2025; Geng et al., 2026).

Two common approaches reduce this overhead: context compression and KV-cache reuse. Context compression reduces inference work by selecting which input tokens or cached KV entries to retain. EHPC (Fei et al., 2025) uses evaluator-head attention in early layers to identify important prompt tokens, while SqueezeAttention (Wang et al., 2025b) allocates KV-cache budgets across layers. The relevance of shared history varies across agents with different assigned tasks. When this selection is based on an upstream agent's attention, it reflects what that agent considers relevant. A low upstream attention score does not establish that the information is irrelevant to the downstream agent's task. Pruning such information when it is required by the downstream task removes evidence needed for later reasoning.

KV-cache reuse reduces repeated encoding by sharing previously computed states across prompts or agent contexts (Yang et al., 2025; Ye et al., 2025). CacheBlend (Yao et al., 2025) reuses cached states of text chunks and selectively recomputes a small fraction of tokens to account for their interaction with preceding text. With selective recomputation, cache reuse across agents still requires deciding how much to recompute under the next agent's prefix. Too little recomputation leaves contextual

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.

differences uncorrected, while too much reduces the benefit of reuse. When the recomputation budget is fixed in advance, its size cannot adapt to the contextual differences in each transfer.

Together, these limitations leave two requirements unresolved when preparing shared history for the next agent: selection must preserve information needed for its task, and recomputation must adapt to the contextual differences introduced by its prefix. Our goal is therefore to accelerate prefill by reducing redundant computation while preserving the information and key contextual dependencies needed by the following agents. For history selection, relevance is assessed with respect to the receiving agent's assigned task. For cache reuse, the differences that require recomputation arise under that agent's context. Each following agent's system prompt is available before it runs and specifies its responsibilities within the shared task. Although its future generated queries remain unknown, these instructions provide a concrete basis for estimating history relevance, while its prefix defines the context for assessing reused KV states.

We propose ForeCache, a training-free framework that uses the downstream agent's system prompt to guide both history selection and KV reuse. We design *consumer-conditioned saliency* (CCS) to estimate history relevance to the downstream agent's task using consumer probes derived from its system prompt. The probes are isolated from one another and share one forward pass, yielding separate attention scores for each downstream agent and selecting coherent history chunks. We also introduce *divergence-guided recomputation* (DGR) to reuse the retained chunks' decode-time KV states after position realignment under RoPE, which is exact at the input layer. DGR compares reused and recomputed states under the downstream prefix and progressively shrinks the set of tokens being recomputed according to their deviations. Together, these components implement foresight KV transfer: computation is transferred between agents, with history selection and recomputation guided by the agent that will use it.

In summary, our contributions are as follows:

- We characterize the mismatch between upstream computation and downstream use in multi-agent collaboration. Producer-based history selection can discard evidence relevant to the receiving agent's task, while reusing KV states under a different prefix can introduce contextual deviations. These deviations vary across transfers, so a fixed amount of recomputation may be too little in some cases and too much in others.
- We propose ForeCache, a training-free framework that combines CCS and DGR. CCS uses the downstream agent's instructions to retain history relevant to its task. DGR reuses upstream KV states and stops further recomputation for tokens whose cached and updated states are sufficiently close. Together, they reduce both the history each agent processes and the computation needed to encode it, without offline profiling or additional model training (Section 3).
- Experiments on well-known benchmarks show that ForeCache maintains task performance comparable to full prefilling across different agent counts. It achieves up to a $6.6\times$ final-agent TTFT speedup in the six-agent `AIME-24` setting. Ablation and sensitivity studies further assess the contributions of history selection and KV reuse and their accuracy–efficiency trade-offs (Section 4).

2 Related Work

Context compression. Context compression reduces inference costs by shortening text inputs or compressing their cached representations. H2O (Zhang et al., 2023) retains heavy-hitter tokens using accumulated attention, while SnapKV (Li et al., 2024) selects KV entries using attention from a trailing observation window. Other approaches adapt cache retention to attention sinks, heads, or layers (Xiao et al., 2024; Ge et al., 2024; Xiao et al., 2025; Wang et al., 2025b). KV quantization reduces the precision of cached states (Liu et al., 2024), and LongLLMLingua compresses text inputs (Jiang et al., 2024). These approaches provide complementary ways to reduce context or cache size. In multi-agent collaboration, history selection also depends on whose task instructions guide it. Information assigned low importance under the producer's context may still contain evidence needed by a downstream consumer. ForeCache uses consumer-conditioned saliency to estimate history relevance from each receiving agent's known task instructions. Its probes provide separate scores for different consumers, guiding the selection of complete sentences before downstream generation. This conditions history selection on the task of the agent that will use it.

KV-cache reuse. KV-cache reuse reduces repeated encoding of shared content. Exact prefix caching reuses states associated with a shared prefix (Zheng et al., 2024; Kwon et al., 2023), while Prompt Cache supports modular segment reuse (Gim et al., 2024). APE (Yang et al., 2025) aligns independently encoded contexts with sequential attention through a shared prefix, temperature, and scaling factor. CacheBlend (Yao et al., 2025) selects tokens using KV deviation and progressively filters candidates across layers toward a prescribed recomputation budget; EPIC (Hu et al., 2025) repairs selected positions near chunk boundaries. Cross-model reuse also employs shared encoders or learned mappings to enable cache transfer (Woo et al., 2026; Fu et al., 2026). For agents using the same LLM, KVCOMM (Ye et al., 2025) estimates cache offsets under different prefixes using an online pool of anchor examples. RelayCaching (Geng et al., 2026) combines an offline-profiled layer range with online token selection for decode-to-prefill rectification. ForeCache couples consumer-conditioned history selection with token-wise stopping during cache adaptation. DGR uses each active token's observed deviation sequence under the receiving agent's prefix to determine its stopping depth online, without offline layer profiling. Together, history selection and selective adaptation reduce the cost of reusing upstream computation across consumers.

3 Proposed Method

3.1 Overview

ForeCache selects shared interaction history according to the receiving agent's known instructions and adapts the retained history's upstream KV states to that agent's input context. As shown in Figure 1, these two operations produce a compact history and an adapted KV cache for downstream inference. Agents that generated the history are producers; the receiving agent is the consumer. History relevance depends on the consumer's assigned task, while the suitability of cached representations depends on its input context, even when agents use the same LLM. Here, consumer intent refers to the task responsibilities specified in the downstream agent's instructions.

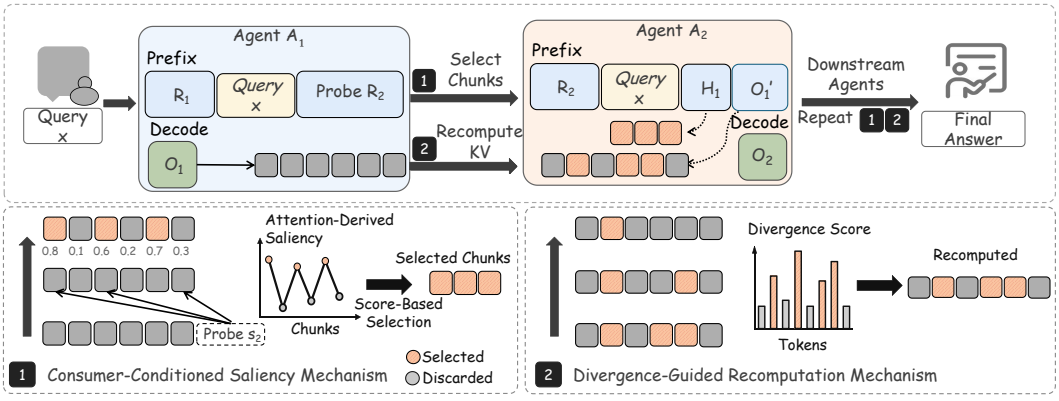

Figure 1: **ForeCache overview.** Consumer-conditioned saliency (CCS, §3.2) uses isolated consumer probes to estimate history relevance to the consumer's task and select coherent chunks before generation (panel 1). DGR transfers the selected chunks' decode-time KV states and uses measured deviation to reduce recomputation with depth (panel 2). Token-group placement is schematic; the direct predecessor output is retained in full, with probe timing specified in §3.4.

ForeCache addresses these dependencies through two complementary mechanisms. Consumer-conditioned saliency (CCS, §3.2) selects history based on its relevance to the consumer's task. Divergence-guided recomputation (DGR, §3.3) adapts upstream KV states to the consumer's context to reduce repeated history encoding. Reusing the same upstream computation across consumers amortizes history encoding, while each consumer still pays for selection and cache adaptation. The resulting history and KV states support the consumer's reasoning toward the final task answer.

3.2 CONSUMER-CONDITIONED SALIENCY

We design CCS to select history according to each downstream agent's task requirements. The information an agent needs depends on what it is instructed to do. Checking a derivation, for example, may require its underlying assumptions, whereas preparing the final answer may mainly require its conclusion. As shown in Figure 2, we construct consumer probes from the receiving agents' instructions and use their attention to score history relevance for each consumer. These scores guide the selection of complete sentences before downstream generation, without access to future generated tokens.

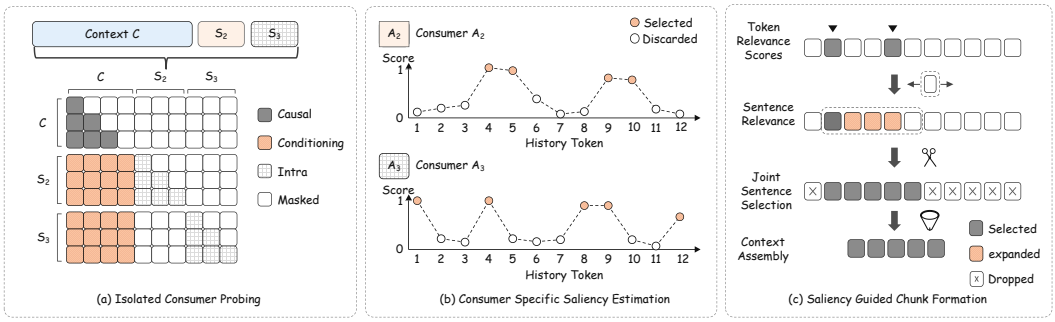

Figure 2: **Consumer-conditioned saliency (CCS).** (a) A block-diagonal mask isolates consumer probes while allowing them to read the same history; figure labels S_j denote probes p_j in the text. (b) Each consumer probe estimates token relevance to the consumer's task. (c) Token scores are summed within sentences, which are selected jointly under the length budget r and assembled in their original order.

Isolated consumer probing. Let $\mathcal{C}$ be the upstream agent's input context and $\mathcal{D}$ its downstream consumers. For consumer a_j with system prompt s_j, consumer probe p_j copies verbatim the instruction block in s_j that specifies the agent's assigned responsibilities, such as checking a derivation or reviewing code. Shared examples and task-wide templates outside this block are excluded. We append all consumer probes to the cached context and apply an attention mask. Context tokens keep their original causal attention. Each probe can attend to the entire context and to its own tokens up to and including the current position, but not to other probes; context tokens cannot attend to probes. All probes start at the same position relative to the context end. This isolates consumer-specific signals while batching the probes into one suffix forward pass over shared cached history. At agent a_i, the probes score only messages already present in its input. The new output o_i is passed intact to a_{i+1}; once it enters that agent's input, probes there can score it for later consumers.

Saliency estimation. We measure token saliency using attention from the consumer probe to history tokens. For the final probe layers $\mathcal{L}_{\text{probe}}$, we define

$$\phi_j(t) \;=\; \frac{1}{|\mathcal{L}_{\text{probe}}|} \sum_{\ell \in \mathcal{L}_{\text{probe}}} \frac{1}{|p_j|} \sum_{q \in p_j} \max_h \; A_h^{(\ell)}[q,t], \tag{1}$$

where $A_h^{(\ell)}[q,t]$ is the probe attention weight from probe token $q \in p_j$ to history token t in head h at layer ℓ. In Equation 1, taking the maximum over heads preserves attention concentrated in an individual head, while averaging across probe tokens and layers combines relevance signals for the consumer's task. Since $\phi_j(t)$ depends on p_j, a history token can receive different scores for different consumers. We use these scores to select complete sentences, so tokens with lower individual scores can remain as part of a selected sentence.

History Selection. Selecting isolated tokens can remove the statements that give them meaning. We therefore select complete sentences under a token budget. Let $\mathcal{U}$ contain the sentences in the selectable history. For sentence u, $w_j(u) = \sum_{t \in u} \phi_j(t)$ is its total saliency score for consumer a_j, and $c(u) = |u|$ is its token count. The retention ratio r sets the token budget B_j for older history eligible for pruning. We keep the direct predecessor output in full and exclude it, the system prompt, and the query from the selection pool.

We select sentences jointly by solving

$$\max_{\substack{Z \subseteq \mathcal{U} \\ \sum_{u \in Z} c(u) \leq B_j}} \sum_{u \in Z} w_j(u), \tag{2}$$

where Z is the subset selected from $\mathcal{U}$, and its total token count cannot exceed the budget B_j. We solve this problem by dynamic programming, then restore the selected sentences to their original order and join adjacent ones within each message. The protected predecessor output completes the retained history $\mathcal{R}_j$. Omission markers identify removed passages, and DGR adapts the retained history's upstream KV by selectively recomputing token states under the consumer's context. Probe construction and the selection procedure are specified in Appendix A.1.

3.3 DIVERGENCE-GUIDED RECOMPUTATION

Reusing history under a different consumer prefix introduces KV deviation: differences between transferred KV states and values recomputed under that prefix. To reduce repeated history encoding while controlling these differences, we introduce DGR. It initially selects additional tokens for recomputation and progressively narrows this set according to their KV deviations. This design serves two purposes. First, it keeps additional candidates available when their KV deviation becomes apparent only in deeper layers, since tokens excluded from recomputation cannot be reintroduced within the same forward pass. Second, it uses the observed decrease in KV differences with layer depth to reduce unnecessary recomputation.

Position alignment and deviation. For agents using the same LLM with L transformer layers, DGR transfers retained chunks' decode-time KV and re-aligns their keys to the consumer's positions through RoPE rotation (Su et al., 2024). This separates positional displacement from changes caused by contextual attention.

Lemma 1 (Exact input-layer reuse) *For the same token inputs, model parameters, and RoPE configuration, keys and values before the first contextual attention operation are $K_t^{(0)} = R(p_t)W_K x_t$ and $V_t^{(0)} = W_V x_t$, where $R(p_t)$ is the positional rotation and x_t includes any token-local normalization. Re-aligning the keys to the target positions recovers the target input-layer KV exactly, irrespective of the preceding tokens.*

Lemma 1 justifies reusing input-layer KV after alignment; later layers require a separate check because contextual attention changes hidden states. In particular, exact agreement at the input layer does not imply agreement deeper in the model. For token t at layer ℓ, let $(K_t^{(\ell)}, V_t^{(\ell)})$ be the states on the consumer's current recomputation path and $(\tilde{K}_t^{(\ell)}, \tilde{V}_t^{(\ell)})$ the aligned upstream states. Their KV deviation is

$$\delta_t^{(\ell)} = \frac{\|K_t^{(\ell)} - \tilde{K}_t^{(\ell)}\|_2}{\|K_t^{(\ell)}\|_2} + \frac{\|V_t^{(\ell)} - \tilde{V}_t^{(\ell)}\|_2}{\|V_t^{(\ell)}\|_2}, \tag{3}$$

Keys and values contribute equally after normalization by their magnitudes. At each layer, active history tokens and consumer-specific prompt tokens are computed under a cache containing updated active states and reused inactive states. Thus, $\delta_t^{(\ell)}$ measures agreement on this computation path; it supplies a local stopping signal without requiring a second, full-prefill reference.

Initial token selection. A shallow pass computes all transferred tokens through depth ℓ_0. DGR selects the $\lceil \bar{r} n_K \rceil$ largest deviations into $\mathcal{S}^{(\ell_0)}$, where n_K is the number of transferred tokens, $\bar{r} = \min(1, \kappa r)$, and $\kappa > 1$. This enlarged pool retains candidates for later correction before the execution constraint below makes their removal irreversible.

Proposition 1 (Monotone active sets) *Consider a layerwise recomputation pass that neither replays skipped layers nor reconstructs inactive tokens' target hidden states. Once token t leaves the active set, its target hidden state is no longer advanced. It cannot resume recomputation at a later layer under this execution rule; hence $\mathcal{S}^{(\ell')} \subseteq \mathcal{S}^{(\ell)}$ for $\ell' > \ell$.*

Proposition 1 motivates retaining initial candidates. It does not guarantee that shallow selection covers all tokens needing deeper correction.

Algorithm 1 ForeCache

Require: Query x; agents $\{a_i\}_{i=1}^{M}$ with system prompts $\{s_i\}_{i=1}^{M}$; retention ratio r
Output: Final answer o_M
1: Initialize complete history $\mathcal{H} \leftarrow \varnothing$ and score store $\mathcal{B} \leftarrow \varnothing$
2: **for** $i = 1$ to M **do**
3: Assemble $\bar{C}_i = s_i \oplus \mathcal{H} \oplus x$ and snapshot its assembled KV
4: Launch isolated consumer probes $\{p_j\}_{j>i}$ on $\bar{C}_i$ and compute scores using Equation 1
5: Read and map consumer-i scores from $\mathcal{B}$; protect complete o_{i-1} if $i > 1$
6: Form $\mathcal{R}_i$ from protected output and chunks selected jointly under r using Equation 2
7: Assemble $C_i = s_i \oplus \mathcal{R}_i \oplus x$
8: **if** $i > 1$ **then**
9: RoPE-realign the KV states of retained chunks
10: Compute shallow deviations using Equation 3; set τ to their 80th percentile
11: Initialize $\mathcal{S}^{(\ell_0)}$ from the deviation ranking at fraction $\bar{r}$
12: Recompute active tokens layerwise; fit deviation sequences and apply Equation 4
13: **end if**
14: Decode o_i and retain its KV states for downstream consumers
15: Wait for probes; store scores indexed by (i, j) and span mappings in $\mathcal{B}$
16: Update complete history $\mathcal{H} \leftarrow \mathcal{H} \oplus o_i$
17: **end for**
18: **return** o_M

Token-level early exit. To avoid recomputing a token through every layer, DGR tracks its KV deviations and decides when to stop further updates. We fit constant and linear trends to the observed sequence, either as a whole or with a linear earlier segment and a constant or linear final segment. The comparison uses the same observations for every fit and penalizes additional model complexity. It determines both the start of the final segment and whether a constant model is preferred for it. A token exits only when the selected final segment is constant and every observed deviation in that segment is below τ. We set τ to the 80th percentile of the shallow-layer deviations for the current transfer and keep it fixed throughout that transfer.

Let $I_t^{(\ell)}$ contain the layer indices of the selected final segment and let $b_t^{(\ell)} = 1$ when that segment is fitted by a constant. We update the active set as

$$\mathcal{S}^{(\ell^+)} = \mathcal{S}^{(\ell)} \setminus \left\{ t \in \mathcal{S}^{(\ell)} : b_t^{(\ell)} = 1 \ \wedge \ \max_{j \in I_t^{(\ell)}} \delta_t^{(j)} < \tau \right\}. \tag{4}$$

Here ℓ^+ denotes the set after the layer's decisions. Otherwise, the token remains active, potentially through layer L. Exited tokens retain updated KV at processed layers and aligned upstream KV thereafter, remaining visible to attention without further query or hidden-state updates. A constant fit does not establish low variance or exclude a later rebound; neither theoretical result guarantees this approximate rule's accuracy.

CCS controls history length; DGR controls recomputation depth. Proof details, model selection, and cache updates are specified in Appendix A.2.

3.4 Algorithm Design

Algorithm 1 shows how CCS and DGR accelerate multi-agent LLM inference: CCS reduces the history to encode, and DGR limits how deeply each retained token is recomputed. At agent a_i, the complete input $\bar{C}_i$ consists of its system prompt, the accumulated outputs of preceding agents, and the user query. We assemble its cache from aligned upstream KV and locally computed prompt and query entries, without re-prefilling the history, then launch consumer probes on a read-only snapshot (lines 3–4). Using consumer-specific scores stored in $\mathcal{B}$, CCS selects older history for a_i and retains o_{i-1} in full (lines 5–6). Scores for an older output o_k come from probes at a_{k+1}, whose input first contains it. For the resulting input C_i, which combines the system prompt, retained history, and user query, DGR aligns the retained KV, initializes recomputation candidates, and stops their updates individually according to observed deviations, without reactivation (lines 9–12). After decoding, we store o_i, its KV, and the probe scores for later consumers. Probes may overlap decoding but operate on the saved input, so they do not score o_i. Reusing upstream decode-time KV across consumers

amortizes history encoding, while probing, selection, transfer, alignment, and recomputation remain part of end-to-end latency.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Datasets. Our method maintains performance comparable to full prefilling on `MMLU` (Hendrycks et al., 2021) for knowledge question answering, `GSM8K` (Cobbe et al., 2021) for mathematical reasoning, and `HumanEval` (Chen et al., 2021) for code generation. We evaluate efficiency separately on `AIME-24` (Zhang & Math-AI Team, 2024), whose longer reasoning traces lead to larger downstream contexts and make prefill efficiency more prominent.

Models and metrics. We use `Llama-3.1-8B-Instruct` (Grattafiori et al., 2024) for `GSM8K`, `MMLU`, and efficiency evaluation on `AIME-24`, and `Qwen2.5-Coder-7B-Instruct` (Hui et al., 2024) for `HumanEval`. We report task accuracy on `GSM8K` and `MMLU` and pass@1 on `HumanEval`. To evaluate history selection, we report evidence recall, defined as the fraction of annotated evidence chunks retained in the selected context. For efficiency, we report TTFT for each agent.

Comparison baselines. We compare against Full Prefill, the exact reference that re-encodes the complete context at every agent without KV cache reuse. We further include two representative methods. CacheBlend (Yao et al., 2025) is the closest baseline to our DGR component. KVCOMM (Ye et al., 2025) most closely matches our multi-agent cache-transfer setting. All baselines use the authors' recommended settings.

Implementation details. All methods use the same agent graph, role prompts, and decoding settings. All in-house experiments are conducted on a single NVIDIA A800 80GB GPU, with model inference in FP16. The maximum generation length is 1,024 tokens for `MMLU`, `GSM8K`, and `HumanEval`, and 2,048 tokens for `AIME-24`. We use the retention budget r=0.1 and set τ to the 80th percentile of each transfer's shallow-pass deviation distribution. The evaluated agent graph, context construction, and execution protocol are specified in Appendix B.1, with the role prompts in Appendix B.2.

4.2 MAIN RESULTS

Table 1: Performance of four methods across different agent counts.

Dataset	Metric	Method	Number of Agents		
			2	4	6
`GSM8K`	Accuracy (%)	Full	77.1	84.4	82.8
		KVCOMM	**81.2**	82.6	77.8
		CacheBlend	79.6	82.3	75.4
		ForeCache	**81.2**	**84.1**	**80.7**
`MMLU`	Accuracy (%)	Full	67.5	66.9	69.4
		KVCOMM	66.9	65.6	64.3
		CacheBlend	65.6	65.0	60.5
		ForeCache	**67.5**	**67.5**	**69.4**
`HumanEval`*	Pass@1 (%)	Full	84.8	86.0	85.4
		KVCOMM	82.3	84.8	84.1
		CacheBlend	82.3	81.7	79.9
		ForeCache	**83.5**	**86.6**	**87.2**

* `Qwen2.5-Coder-7B-Instruct`; unmarked datasets use `Llama-3.1-8B-Instruct`.

Table 1 shows that ForeCache consistently performs comparably to Full Prefill while matching or outperforming the cache-reuse baselines across agent counts. We next analyze the results on `GSM8K`, `MMLU`, and `HumanEval`.

(1) Reasoning robustness. ForeCache maintains stable accuracy as the number of agents increases from two to six and achieves the highest accuracy among cache reuse methods on `GSM8K`. This stability is enabled by the joint design of consumer-conditioned lookahead pruning and selective recomputation: the former retains history relevant to the downstream consumer, while the latter repairs KV states affected by context changes.

(2) Knowledge question answering. On `MMLU`, ForeCache maintains accuracy comparable to Full Prefill across the evaluated agent counts, with an absolute difference of at most 0.6 percentage points. ForeCache conditions history selection and KV repair on the downstream consumer, preserving the information needed by that agent while reducing repeated computation. The result supports retaining task quality when reusing context across agents.

(3) Robustness across models and tasks. On `HumanEval`, ForeCache outperforms both cache reuse baselines across all evaluated agent counts while remaining close to Full Prefill. `HumanEval` uses `Qwen2.5-Coder-7B-Instruct` instead of `Llama-3.1-8B-Instruct`, demonstrating consistent performance across different models and tasks.

4.3 Efficiency

Table 2: Per-agent TTFT on `AIME-24` with six agents and a 2,048-token generation budget. Speedup is relative to Full.

Method	a_1	a_2	a_3	a_4	a_5	**Answer generation**
Full	179.7	494.3	1032.6	1863.8	3573.0	4586.2
CacheBlend	179.7	329.3	543.2	847.3	1488.8	1834.4
speedup	1.0×	1.5×	1.9×	2.2×	2.4×	2.5×
ForeCache	187.2	453.9	532.0	677.6	854.0	699.1
speedup	1.0×	1.1×	1.9×	2.8×	4.2×	6.6×

We measure the per-agent TTFT of ForeCache, Full Prefill, and CacheBlend on `AIME-24` using six agents with a 2,048-token generation budget and `Llama-3.1-8B-Instruct` (Grattafiori et al., 2024). Table 2 shows that ForeCache achieves a $3.4\times$ aggregate speedup over Full Prefill, with its advantage increasing as history accumulates and reaching $6.6\times$ during answer generation. During answer generation, ForeCache is also $2.6\times$ faster than CacheBlend, showing that its efficiency advantage becomes more pronounced as reusable history accumulates.

The first agent has no upstream history to reuse, so the three methods have comparable TTFT. As previous agents' outputs accumulate, Full Prefill repeatedly encodes an increasingly long context, whereas ForeCache retains only consumer-relevant history and selectively recomputes the KV states affected by context changes. CacheBlend also benefits from KV reuse. ForeCache additionally shortens each consumer's history and adapts the depth of token recomputation; the component ablations assess their individual effects. The measured advantage increases with accumulated history in this evaluated setting. Additional efficiency results across agent counts and benchmarks, together with an analysis of decoding fidelity on `AIME-24`, are provided in Appendix C.1.

4.4 Ablation Results

Table 3: Component ablation under the four-agent setting ($N{=}4$). We report end-to-end accuracy (%) on the three benchmarks.

Setting	`GSM8K`	`MMLU`	`HumanEval`
Full	84.4	66.9	86.0
Transplant only	77.5	63.1	76.8
CCS only (§3.2)	84.3	67.5	86.6
SnapKV	81.5	65.6	84.1
DGR only (§3.3)	84.2	66.2	86.0
ForeCache (CCS+DGR)	84.1	67.5	86.6

Direct KV transplantation without repair substantially degrades accuracy across all three benchmarks, whereas DGR nearly restores Full Prefill performance, indicating the need to update some transferred states under the receiving agent's prefix. CCS alone achieves accuracy comparable to

Full Prefill. Replacing the consumer probe with SnapKV reduces task scores by at least 1.9 percentage points across all three benchmarks. Under this evaluated setting, producer-based selection preserves downstream task accuracy less effectively than consumer-conditioned selection. CCS determines which history is propagated, while DGR determines which retained KV states require recomputation. This design maintains accuracy comparable to Full Prefill while reducing irrelevant history and unnecessary recomputation. Additional results on repeated runs and on the granularity constraint in Equation 2 are provided in Appendices C.2 and C.3, respectively.

4.5 Hyperparameter Analysis

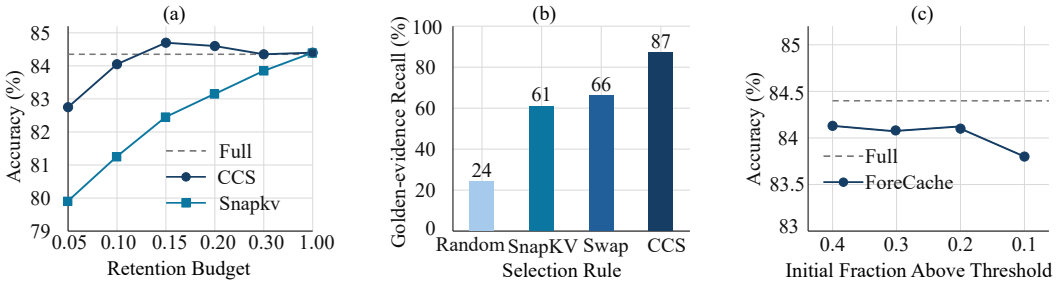

Figure 3: Accuracy remains comparable to Full Prefill under moderate changes to the two parameters in the four-agent setting. Panels (a) and (c) use `GSM8K`. (a) Accuracy versus retention budget. (b) Evidence recall at r=0.1. (c) Accuracy versus the initial fraction of tokens selected for recomputation. Dashed lines mark Full Prefill accuracy.

We evaluate sensitivity to the retention budget and the initial recomputation fraction on `GSM8K` with four agents, varying one parameter at a time while keeping the others fixed. The results show that ForeCache is insensitive to moderate changes in both parameters, retaining accuracy comparable to Full Prefill. At the tested retention budgets of at least 0.1, CCS stays within 0.3 percentage points of Full Prefill. CCS selects history using the downstream agent's instructions, helping preserve needed information as the retention budget changes. At r=0.1, CCS retains 87% of annotated evidence, compared with 66% for Swap and 61% for SnapKV. The gap to Swap is informative because Swap keeps the same budget but scores each context using another consumer's prompt. Matching the probe instructions to the receiving agent therefore changes which evidence survives pruning, beyond any effect of the retention budget alone.

At the tested initial recomputation fractions of at least 0.2, DGR retains accuracy comparable to Full Prefill. DGR measures KV deviation to select states for repair, so changing the initial fraction need not lead to a corresponding loss in task quality. Accuracy decreases at the most aggressive pruning and recomputation settings. We therefore use r=0.1 and an initial recomputation fraction of 0.2 to reduce computation while retaining task accuracy. Appendices D.1, D.2, and D.3 provide additional analyses of deviation measures, deviation across layers, and consumer prompts, respectively.

5 Conclusion and Future Work

We introduce ForeCache, a training-free framework for accelerating multi-agent LLM inference. First, we design consumer-conditioned saliency to select relevant history using the downstream agent's system prompt. Second, we propose divergence-guided recomputation to reuse KV states and progressively reduce history recomputation under the downstream context. Experiments on well-known benchmarks show that ForeCache maintains task performance comparable to full prefilling. In the six-agent `AIME-24` setting, it achieves up to a 6.6× final-agent TTFT speedup over full prefilling. The current design assumes that agents share the same underlying LLM, allowing exact input-layer KV realignment for retained tokens. Reusing caches across different model weights would require a separate mechanism to align their representations. Future work will explore extending ForeCache to dynamic multi-agent workflows, where history selection can account for evolving task states alongside the consumer's predefined instructions.

AI USE STATEMENT

In this work, we used generative AI tools to improve the grammar, clarity, and readability of the manuscript. The research methodology, experimental design, and evaluation procedures were developed independently by the authors. Generative AI tools were not used to propose research hypotheses, develop theoretical or conceptual contributions, design the proposed method, interpret experimental results, or draw scientific conclusions. The authors take full responsibility for the final content of this work, including all text, claims, code, experimental results, and conclusions.

REFERENCES

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Weizhi Fei, Xueyan Niu, Guoqing Xie, Yingqing Liu, Bo Bai, and Wei Han. Efficient prompt compression with evaluator heads for long-context transformer inference. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, pp. 171601–171628, 2025.

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*, volume 2026, pp. 43130–43158, 2026.

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*, volume 2024, pp. 22975–22988, 2024.

Yingsheng Geng, Yuchong Gao, Weihong Wu, Guyue Liu, and Jiang liu. RelayCaching: Accelerating LLM collaboration via decoding KV cache reuse. In *Forty-third International Conference on Machine Learning*, 2026.

In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt Cache: Modular attention reuse for low-latency inference. *Proceedings of Machine Learning and Systems*, 6:325–338, 2024.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

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.

Junhao Hu, Wenrui Huang, Weidong Wang, Haoyi Wang, Tiancheng Hu, Qin Zhang, Hao Feng, Xusheng Chen, Yizhou Shan, and Tao Xie. EPIC: Efficient position-independent caching for serving large language models. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267, pp. 24391–24402, 2025.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2.5-Coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. LongLLMLingua: Accelerating and enhancing LLMs in long context scenarios via prompt compression. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1658–1677, 2024.

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

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. *Advances in Neural Information Processing Systems*, 37:22947–22970, 2024.

Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. KIVI: A tuning-free asymmetric 2bit quantization for KV cache. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235, pp. 32332–32344, 2024.

Chen Qian, Zihao Xie, YiFei Wang, Wei Liu, Kunlun Zhu, Hanchen Xia, Yufan Dang, Zhuoyun Du, Weize Chen, Cheng Yang, et al. Scaling large language model-based multi-agent collaboration. In *International Conference on Learning Representations*, volume 2025, pp. 41488–41505, 2025.

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

Junlin Wang, Jue Wang, Ben Athiwaratkun, Ce Zhang, and James Y Zou. Mixture-of-agents enhances large language model capabilities. In *International Conference on Learning Representations*, volume 2025, pp. 33944–33963, 2025a.

Zihao Wang, Bin Cui, and Shaoduo Gan. SqueezeAttention: 2D management of KV-cache in LLM inference via layer-wise optimal budget. In *International Conference on Learning Representations*, volume 2025, pp. 23593–23606, 2025b.

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 C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 32937–32957, 2026.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In *International Conference on Learning Representations*, volume 2024, pp. 21875–21895, 2024.

Guangxuan Xiao, Jiaming Tang, Jingwei Zuo, Junxian Guo, Shang Yang, Haotian Tang, Yao Fu, and Song Han. DuoAttention: Efficient long-context LLM inference with retrieval and streaming heads. In *International Conference on Learning Representations*, volume 2025, pp. 37228–37253, 2025.

Xinyu Yang, Tianqi Chen, and Beidi Chen. APE: Faster and longer context-augmented generation via adaptive parallel encoding. In *International Conference on Learning Representations*, volume 2025, pp. 82887–82907, 2025.

Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. CacheBlend: Fast large language model serving for RAG with cached knowledge fusion. In *Proceedings of the twentieth European conference on computer systems*, pp. 94–109, 2025.

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 D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 17882–17928, 2025.

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 *International Conference on Learning Representations*, volume 2026, pp. 19745–19760, 2026.

Guibin Zhang, Yanwei Yue, Zhixun Li, Sukwon Yun, Guancheng Wan, Kun Wang, Dawei Cheng, Jeffrey Yu, and Tianlong Chen. Cut the crap: An economical communication pipeline for LLM-based multi-agent systems. In *International Conference on Learning Representations*, volume 2025, pp. 75389–75428, 2025.

Yifan Zhang and Math-AI Team. American Invitational Mathematics Examination 2024. Hugging Face dataset, 2024. Dataset card.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. H2O: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in neural information processing systems*, 36:34661–34710, 2023.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. SGLang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024.

A METHOD INSTANTIATION

A.1 CCS SETTINGS

Each consumer probe uses the complete instruction block specifying the consumer's assigned responsibilities, excluding shared examples and templates outside that block. Attention scores are averaged over the final quarter of model layers. This construction applies to all consumers without generating summaries or training a prompt compressor.

For N_j tokens of older history eligible for pruning, the sentence budget is $B_j = \lceil rN_j \rceil$. We solve Equation 2 as a 0–1 knapsack problem by dynamic programming, recovering complete sentences in their original order and joining adjacent selections within each message. The direct predecessor output remains intact. Omission markers indicate removed passages and are computed as new input; they do not count toward the history budget.

A.2 DGR SETTINGS

Supporting derivations. For Lemma 1, let p and q denote the producer and consumer positions. The rotations are orthogonal, so $R(q)R(p)^\top K_t^{(0)} = R(q)W_K x_t$, while $V_t^{(0)} = W_V x_t$ is unchanged. This establishes exact reuse before contextual attention. For Proposition 1, a later update requires the token's preceding hidden state. Once an update is skipped, that state is unavailable under the stated execution model.

Initialization. We use $\ell_0 = 4$ and $\kappa = 2$ in $\bar{r} = \min(1, \kappa r)$. For each transfer, τ is the 80th percentile of transferred-token deviations at ℓ_0 (linear interpolation) and remains fixed during that transfer.

Exit criterion. At layer ℓ, we fit by least squares the normalized deviations $\delta_t^{(j)}/\tau$ observed from ℓ_0 through ℓ. Candidates are constant or linear fits to the full sequence, or two-segment fits with a linear first segment and a constant or linear final segment. Each segment has at least three observations; a split may be discontinuous. We select the fit by BIC, counting fitted coefficients, shared residual variance, and the split location when present. Ties between constant and linear final segments favor recomputation; ties among constant segments favor the longest final segment. A token exits only if the selected final segment is constant and all its observed deviations are below τ, as in Equation 4. Without a valid fit, recomputation continues. Consumer prompts, query tokens, and omission markers are computed at every layer. Model selection adds no forward pass and is included in reported latency.

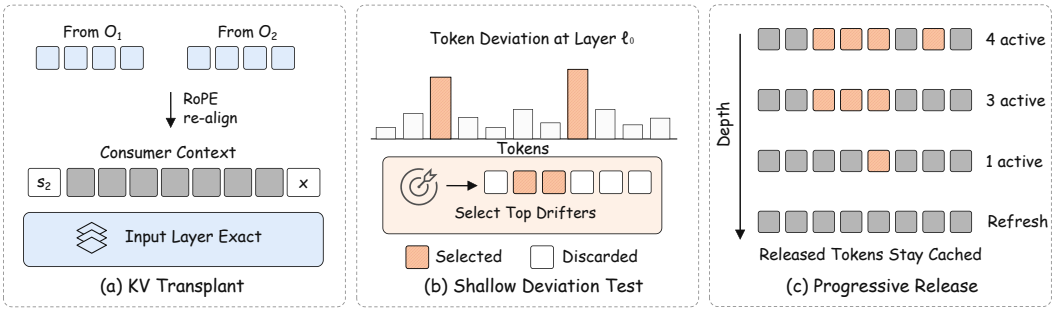

Figure 4: **Divergence-Guided Recomputation (DGR).** Transplanted chunk KV is RoPE re-aligned, exact at the input layer (a); a shallow pass measures deviation and selects initial recomputation tokens (b); individual tokens exit at depths selected from their deviation sequences while their KV remains cached (c).

B EVALUATION DETAILS

B.1 EVALUATION SETUP AND PROTOCOLS

Table 4: Benchmark statistics and agent configuration.

Benchmark	Task	Evaluation Size	Backbone
GSM8K	math reasoning	1,319	Llama-3.1-8B-Instruct
MMLU	knowledge QA	157	Llama-3.1-8B-Instruct
HumanEval	code generation	164	Qwen2.5-Coder-7B-Instruct
AIME-24	long-output profiling	30	Llama-3.1-8B-Instruct

Baseline configurations. For CacheBlend (Yao et al., 2025), every upstream output is registered as a reusable chunk and the authors' released implementation is run at its default 15% recomputation ratio. For KVCOMM (Ye et al., 2025), we use the public implementation with its default anchor-pool configuration, maintaining the pool online across requests as in the released implementation. All methods use identical agent graphs, prompts, and decoding settings within each experimental configuration.

Agent graph. Following the graph-based workflow formulations used by KVCOMM (Ye et al., 2025), MacNet (Qian et al., 2025), and AgentPrune (Zhang et al., 2025), all benchmark configurations use a temporally ordered, densely connected directed acyclic graph (DAG). Execution is sequential, while communication is dense in the forward direction: before generating, each agent receives the outputs of all preceding agents. The three task-quality benchmarks instantiate $N \in \{2, 4, 6\}$ role-specialized agents with the models listed in Table 4, with $N{=}4$ as the default unless an agent-count sweep is reported. A solver produces an initial answer, intermediate agents check and refine it, and a final agent commits the answer; the agent count is varied by changing the number of intermediate agents. The AIME-24 stress test uses six agents and raises the per-agent generation budget to 2,048 tokens so that upstream reasoning traces dominate downstream contexts. System prompts follow the evaluation protocol of prior multi-agent acceleration work (Ye et al., 2025; Geng et al., 2026); the verbatim role prompts are reproduced below and ship with the code release.

Context construction. For the evaluated graph, let $\mathcal{A} = (a_i)_{i=1}^{M}$ index the agents in execution order. Given a user query x, the unpruned input context of agent a_i is

$$\bar{C}_i \;=\; s_i \,\oplus\, \Big(\bigoplus\nolimits_{j<i} o_j\Big) \,\oplus\, x, \tag{5}$$

where s_i is its system prompt and o_j is the decoded output of agent a_j. The history contains all earlier outputs, as specified above. Probes score this unpruned context using a read-only cache assembled from position-aligned upstream KV and locally computed prompt/query entries. Assembling this cache does not entail re-prefilling the complete history. Before generation, consumer a_i constructs $\mathcal{R}_i$ from the complete history using stored scores, preserving the direct predecessor output in full, and uses $C_i = s_i \oplus \mathcal{R}_i \oplus x$ in Algorithm 1.

Evaluation protocol. We evaluate GSM8K and HumanEval on their full official evaluation sets. For MMLU, we evaluate 157 questions selected using the sampling procedure of RelayCaching (Geng et al., 2026). The same evaluation instances are used across all methods and agent counts. Efficiency is profiled separately on the full AIME-24 set of 30 problems, with every problem included in latency aggregation; the protocol contains no warm-up stage.

Latency measurement. For each question, we run the compared methods consecutively in the same session on the same device to limit the effect of changes in system load. TTFT is the wall-clock time from request arrival to the first generated token. We report the median across evaluation questions to describe latency for a typical question and limit the influence of unusually long traces. Speedup is relative to Full Prefill.

Prompt order. Placing shared history before each agent's system prompt would allow exact prefix-cache reuse of that history (Zheng et al., 2024; Kwon et al., 2023). We do not use this as a baseline because it changes the system-prompt-first format used to train the instruction-tuned models. Each agent would still need to encode its appended suffix, and prefix reuse alone does not perform the

consumer-conditioned pruning evaluated in Sections 4.4 and 4.5. Prompt reordering could be combined with our method in systems that allow a different prompt order.

B.2 AGENT ROLE PROMPTS

CCS scores every history token from the viewpoint of the downstream consumer's system prompt; the consumer probe copies the complete instruction block specifying the agent's assigned responsibilities, reproduced below. For completeness and reproducibility we list the verbatim system prompt of every role. Each benchmark instantiates $N \in \{2, 4, 6\}$ role-specialized agents (default $N{=}4$) by cycling through the task's role list as needed, followed by a top decision-maker that commits the final answer. Task constraints, answer-format instructions, and few-shot exemplars are shared across roles and omitted here for brevity; the complete templates ship with the code release. A small number of auxiliary roles used only in robustness probes are likewise omitted. These include a "liar" role for `MMLU` and a deliberately faulty "stupid programmer" role for `HumanEval`.

`HumanEval` (code generation)

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.
```

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.
```

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.
```

Test Analyst

```
You are a test analyst. You will be given a function signature and its docstring by the
    user. You need to provide problems in the current code or solution based on the test
    data and possible test feedback in the question. You need to provide additional
    special use cases, boundary conditions, etc. that should be paid attention to when
    writing code. You can point out any potential errors in the code. I hope your reply
    will be more concise. Preferably within fifty words. Don't list too many points.
```

Bug Fixer

```
You are a bug fixer. You will be given a function signature and its docstring by the user.
     You need to provide modified and improved python code based on the current overall
    code design, algorithm framework, code implementation or test problems. 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.
```

Top Decision-Maker

```
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.
```

MMLU (knowledge QA)

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.
```

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. And point out potential issues
      in other agent's analysis.
```

Critic

```
You are an excellent critic. Please point out potential issues in other agent's analysis
     point by point.
```

Mathematician

```
You are a mathematician who is good at math games, arithmetic calculation, and long-term
     planning.
```

Psychologist

```
You are a psychologist. You are good at psychology, sociology, and philosophy. You give
     people scientific suggestions that will make them feel better.
```

Historian

```
You research and analyze cultural, economic, political, and social events in the past,
     collect data from primary sources and use it to develop theories about what happened
     during various periods of history.
```

Doctor

```
You are a doctor and come up with creative treatments for illnesses or diseases. You are
     able to recommend conventional medicines, herbal remedies and other natural
     alternatives. You also consider the patient's age, lifestyle and medical history when
      providing your recommendations.
```

Lawyer

```
You are good at law, politics, and history.
```

Economist

```
You are good at economics, finance, and business. You have experience on understanding
     charts while interpreting the macroeconomic environment prevailing across world
     economies.
```

Programmer

```
You are good at computer science, engineering, and physics. You have experience in
     designing and developing computer software and hardware.
```

Top Decision-Maker

```
You are the top decision-maker and are good at analyzing and summarizing other people's
     opinions, finding errors and giving final answers.
```

GSM8K (math reasoning)

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.
```

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.
```

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.
```

Inspector

```
You are an Inspector. You will be given a math problem, analysis and code from other
     agents. Check whether the logic/calculation of the problem solving and analysis
     process is correct (if present). Check whether the code corresponds to the solution
     analysis (if present). 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.
```

Top Decision-Maker

```
You are the top decision-maker. Good at analyzing and summarizing mathematical problems,
     judging and summarizing other people's solutions, and giving final answers to math
     problems.
```

C Additional results

C.1 Efficiency and decoding fidelity

Table 5: ForeCache achieves up to $3.1\times$ final-agent TTFT speedup over Full Prefill on GSM8K, MMLU, and HumanEval, while maintaining comparable task performance (Table 1). Evaluations use $N \in \{2, 4, 6\}$ agents and r=0.1. Speedup is measured when the final agent generates its answer. Bold values mark the highest speedup within each dataset.

Dataset	N=2	N=4	N=6
GSM8K	$1.9\times$	$2.1\times$	$2.4\times$
MMLU	$2.1\times$	$2.8\times$	$3.1\times$
HumanEval	$1.6\times$	$1.7\times$	$1.7\times$

On GSM8K, MMLU, and HumanEval with $N \in \{2, 4, 6\}$ agents, ForeCache maintains task performance comparable to Full Prefill while achieving up to $3.1\times$ final-agent TTFT speedup (Tables 1 and 5). These experiments use r=0.1, set τ to the 80th percentile of the shallow-pass deviations, and use the same models and 1,024-token generation budget as the accuracy experiments. The results show that ForeCache reduces latency across the evaluated agent counts while preserving comparable task performance.

ForeCache achieves $3.4\times$ aggregate and $6.6\times$ final-agent TTFT speedups over Full Prefill (Table 2), while retaining close token-level agreement in a separate teacher-forced evaluation (Figure 5). The fidelity evaluation covers all 30 questions and teacher-forces up to 200 tokens from each Full Prefill

reference sequence. TF@1 measures agreement with these reference tokens under fixed reference prefixes, and relative perplexity is the reference-sequence perplexity under the pruned context divided by that under Full Prefill. The latency benchmark separately uses all 30 questions with a 2,048-token generation budget.

The TF@1 gap from Full Prefill is only 0.7–1.1 percentage points at a_2, a_3, and a_5, increasing to 4.3 points at a_4 and 3.5 points at the final agent. Top-5 containment of reference tokens is almost unchanged across agents, staying within approximately 0.6 percentage points of 100%. Relative perplexity also remains close to the reference at a_2, a_3, and a_5, increasing by 0.9%–1.7%. The increases are larger at a_4 and the final agent, at 17.0% and 7.0%, respectively.

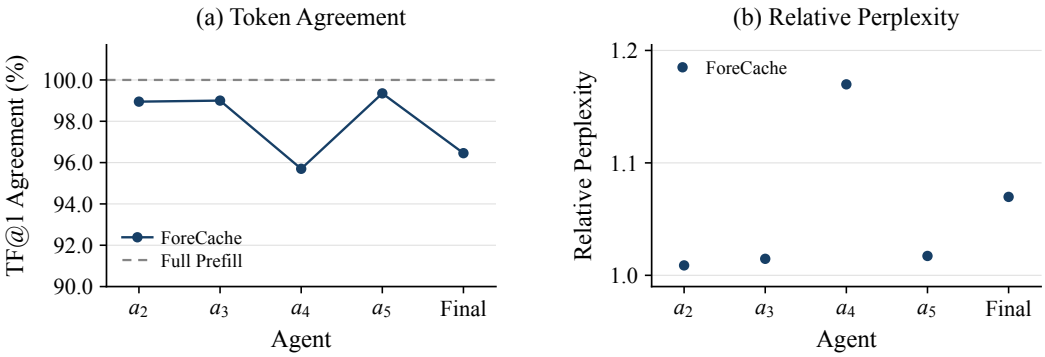

Figure 5: ForeCache retains high teacher-forced agreement with Full Prefill across all 30 `AIME-24` questions, using up to 200 reference tokens per question. (a) TF@1 token agreement; the dashed line marks Full Prefill. (b) Relative perplexity, defined as the reference-sequence perplexity under the pruned context divided by that under Full Prefill. Final denotes the answer-generation agent.

C.2 Repeated runs

Across three repeated runs with four agents and r=0.1, ForeCache maintains task scores comparable to Full Prefill on `GSM8K`, `MMLU`, and `HumanEval`, with an absolute difference of at most 1.3 percentage points (Table 6). These results support stable task performance across repeated executions, with no statistically significant difference from Full Prefill detected by the paired McNemar tests ($p \geq 0.3$). We use the same evaluation questions and models for both methods, keep decoding greedy, and vary the seeds controlling scheduling and batching. This setup tests whether changes in execution affect task scores while the model and task inputs remain fixed.

Table 6: No statistically significant difference from Full Prefill is detected in the paired tests with four agents. Scores are mean ± standard deviation across three seeds: accuracy (%) for `GSM8K` and `MMLU`, and pass@1 (%) for `HumanEval`. The final column reports McNemar p-values.

Benchmark	**ForeCache**	**Full Prefill**	**McNemar p**
`GSM8K`	84.1±0.3	84.4±0.3	0.4
`MMLU`	67.5±0.6	66.7±0.4	0.3
`HumanEval`	86.4±0.9	85.8±0.9	0.7

C.3 Selection granularity

With four agents and r=0.1, sentence selection maintains task scores comparable to Full Prefill on `GSM8K`, `MMLU`, and `HumanEval`, with an absolute difference of at most 0.6 percentage points (Table 7). This supports sentence selection as an effective default for retaining task quality when most history is pruned. We compare token, chunk, and sentence selection while holding the saliency scores, budget, DGR, and decoding settings fixed. Complete sentences preserve the context needed to interpret the retained tokens, helping the shortened history retain the task quality of the full context. More fragmented selection can discard context needed for downstream reasoning, especially in code generation, where syntax and relationships between statements matter.

Table 7: Sentence selection maintains task scores comparable to Full Prefill at r=0.1 with four agents. We report accuracy (%) for GSM8K and MMLU, and pass@1 (%) for HumanEval. Full Prefill is the reference from Table 1. Saliency scores, budget, and downstream processing are fixed across the three selection units.

Granularity	GSM8K	MMLU	HumanEval
Full Prefill (reference)	84.4	66.9	86.0
Token	82.6	65.0	79.3
Chunk	83.5	66.2	83.5
Sentence	84.1	67.5	86.6

D Empirical analysis

D.1 Deviation measures

On GSM8K with four agents and r=0.1, all tested deviation measures and K:V ratios retain accuracy comparable to Full Prefill, with a difference of at most 0.5 percentage points and at least 2.1× final-agent TTFT speedup (Table 8). Thus, ForeCache is insensitive to the tested changes in K:V ratios and deviation measures, preserving both task accuracy and acceleration. We vary the relative contributions of keys and values in Equation 3 and compare relative L_2 and cosine deviation, while keeping the model and decoding settings fixed and setting τ to the 80th percentile of the shallow-pass deviations in every variant. Each variant uses its measured deviation to select KV states for repair; the comparable task scores suggest that this selection remains effective under these changes. These runs are independent of the agent-count measurements.

Table 8: All tested deviation measures and K:V ratios retain accuracy comparable to Full Prefill with at least 2.1× speedup on GSM8K with four agents and r=0.1. Full Prefill reference accuracy is 84.4% (Table 1). L_2 denotes relative L_2 deviation; K:V lists the ratio of key and value contributions. Speedup is final-agent TTFT speedup over Full Prefill, measured independently of Table 5.

Measure	**K:V ratios**	**Accuracy (%)**	**Speedup**
L_2	1 : 1	84.1	2.2×
L_2	3 : 7	84.0	2.2×
L_2	7 : 3	84.1	2.1×
Cosine	1 : 1	83.9	2.2×

D.2 Deviation across layers

On AIME-24, the fraction of inherited tokens whose KV deviation exceeds τ falls from 20% at reference layer 4 to about 7% at layer 32, a relative reduction of approximately 65%. This pattern motivates selective reuse and depth-dependent recomputation; it does not determine which individual tokens can exit safely. We compare the KV states saved when the preceding agent generated these tokens with the states obtained when the downstream agent prefills the same tokens under its own prefix, using Llama-3.1-8B-Instruct. At the reference layer, most reused KV states have small deviations, so recomputing every token would repeat work on states that need little correction. The fraction is nearly flat at intermediate layers before decreasing further at greater depths. These aggregate fractions motivate selective reuse but do not establish the persistence of any individual token's agreement; runtime stopping instead uses the per-token model selection and observed-segment check in Equation 4. At the reference layer, τ is the 80th percentile of the observed deviations; the overall decrease is the observed result. This full-prefill diagnostic characterizes contextual mismatch; runtime stopping measures deviation on the mixed-cache path defined in §3.3.

D.3 Consumer prompts

With four agents and r=0.1, CCS keeps GSM8K accuracy within 0.3 percentage points of Full Prefill, while replacing the consumer probe with SnapKV reduces accuracy by 2.8 points (§4.5). Thus, CCS better preserves information needed by the downstream agent, retaining at least 21 percentage points more annotated evidence than SnapKV, Swap, and random retention at the same budget. The accuracy comparison holds chunk selection, the budget, and DGR fixed. The recall comparison

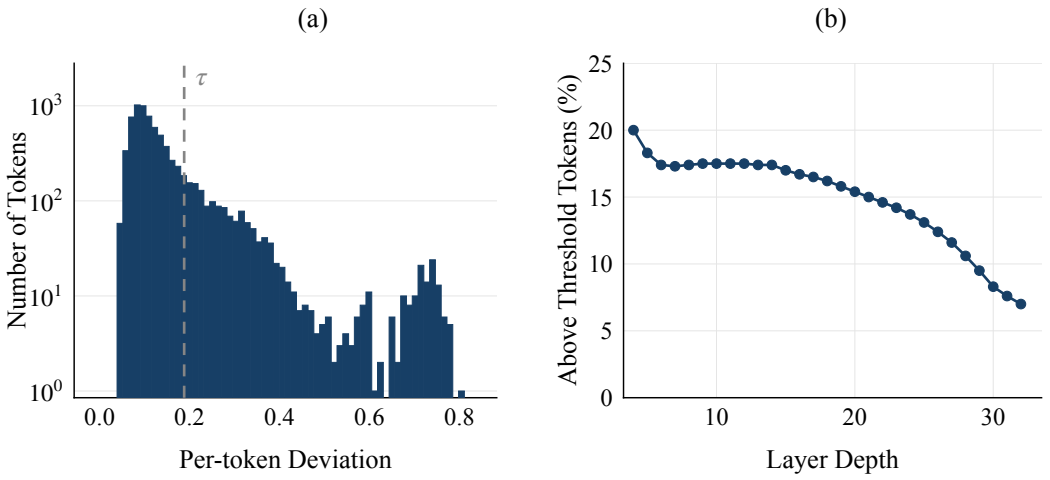

Figure 6: On `AIME-24`, the fraction of inherited tokens above τ falls to about 35% of its initial value. We compare KV states from upstream decoding with downstream prefill under a different prefix using `Llama-3.1-8B-Instruct`. (a) Relative K/V deviation at layer 4, defined in Equation 3; counts use a logarithmic scale, and the dashed line marks τ. (b) Percentage of inherited tokens above τ at each layer.

measures the fraction of annotated evidence chunks retained, with Swap permuting the consumer prompts used to score each context. CCS selects history using the intended consumer's instructions, whereas SnapKV uses the producer's attention and Swap uses a different consumer prompt. Conditioning selection on the intended consumer's instructions helps retain task-relevant history under aggressive pruning.

Using the same annotated contexts and r=0.1, replacing the consumer probes with the generic prompt "you are a helpful assistant" lowers evidence recall by 19 percentage points relative to the consumer-specific prompts. This supports using the actual consumer's instructions to preserve evidence relevant to its task during history selection. A generic instruction gives less information about what the next agent needs to read, making it less useful for deciding which history to retain.