COREMEM: GRAPH-ORGANIZED MEMORY FOR LONG-HORIZON MULTI-AGENT COLLABORATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Long-horizon multi-agent systems distribute planning, retrieval, tool execution, and reflection across many roles and interaction rounds. Their histories are nevertheless often stored as separate Agent conversations, summaries, or vector memories. The same event can therefore be duplicated across private stores, different Agents can act on inconsistent versions of shared state, and an outdated intermediate conclusion can continue to influence later decisions. We introduce CoReMem, a graph-organized memory framework that represents Agents, tasks, subgoals, messages, tool outputs, environment states, and deliverables as typed nodes connected by delegation, reference, dependency, conflict, and temporal edges. Before acting, each Agent retrieves a budgeted local memory subgraph conditioned on its current subtask instead of reading the complete history. When an environment state, tool result, or task plan changes, CoReMem versions the updated node, propagates stale status through dependency edges, and rereads, recomputes, or replans only the affected subgraph before restoring current state. All memory organization and consistency maintenance remain external to the frozen language models. Our evaluation protocol tests the framework on long-horizon data analysis, software construction, and evidence synthesis workflows.

1 INTRODUCTION

Long-horizon multi-agent tasks distribute planning, retrieval, tool execution, and reflection across dozens or hundreds of interaction rounds. Frameworks such as AutoGen and MetaGPT organize these processes through role specialization and message passing (Wu et al., 2024; Hong et al., 2024). Yet their expanding collaboration history creates a persistent systems problem: memory becomes fragmented across Agents while the underlying task state continues to change.

Fragmentation produces three coupled failures. First, multiple Agents may copy the same tool output or intermediate conclusion into conversations and summaries, creating redundant records that can later diverge. Second, the team lacks a state anchor: a planning Agent may believe that a dataset has been cleaned while an execution Agent still reads the raw version. Third, an upstream change can leave "zombie" information in downstream memory. When a source is withdrawn, a tool result is corrected, or a plan is revised, the system must determine which Agent consumed the old value, which conclusions depend on it, and which artifacts are no longer current. Without this information, later decisions can remain internally coherent yet refer to an obsolete task state.

Long contexts and vector retrieval partly address these failures. Full-history prompting preserves information but makes every Agent process growing amounts of unrelated text. Per-Agent summaries reduce context but replicate shared events and hide provenance. Shared vector memory retrieves similar items, yet semantic proximity does not encode operational dependence: a final table may depend on a corrected source even when their text is dissimilar. The memory question in long collaboration is therefore not only *what past content resembles the current query*, but also *what state this Agent needs now, where it came from, and whether it remains valid*.

Graph-structured memory provides useful building blocks for this problem. G-Memory organizes multi-agent experience through interaction, query, and insight graphs (Zhang et al., 2025). MRAgent reconstructs relevant evidence through active graph traversal (Ji et al., 2026), while MAGMA, StructMem, and GAM use relational, event, temporal, or hierarchical structure for long-term retrieval and consolidation (Jiang et al., 2026; Xu et al., 2026; Wu et al., 2026). Provenance-aware

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.

shared memory further records version history and affected lineage (Wang et al., 2026; Margalit et al., 2026). These methods establish that graph structure can preserve relations that flat memory loses. We study a complementary system question: how can one graph serve as the shared memory substrate for an ongoing multi-agent task, provide a different local view to each subtask, and support localized recovery when the represented state changes?

This motivates our central research question: *How can a multi-agent system organize growing collaboration history so that each Agent reads only the state it needs, while an upstream change revises only the downstream work that truly depends on it?*

Our design principle is that long-horizon collaboration should operate on a *shared, versioned dependency state*, not a collection of Agent-owned transcripts. This state must support four operations. It must store a shared event once while preserving who produced and consumed it; retrieve the local dependency context required by the Agent; propagate a state change to memories that depend on the changed node; and recover only the affected work while retaining stable branches. If storage remains private, retrieval cannot establish a common state. If retrieval ignores dependency, it may return obsolete content. If updates do not propagate, stale conclusions survive. If every change triggers full replay, the system loses the efficiency gained by structured memory.

We introduce CoReMem, a graph-organized memory framework for this lifecycle. CoReMem maps Agents, tasks, subgoals, messages, tool outputs, environment snapshots, and deliverables to typed nodes. Delegation, reference, dependency, conflict, and temporal relations become typed edges. Canonical node identities allow Agents to point to the same event rather than storing independent copies, while node metadata records producer, consumer, verification status, version, and validity.

Before an Agent acts, CoReMem constructs a query from the Agent identity, current subtask, active plan, and immediate observation. A lightweight graph retriever finds semantic seeds, expands them along task and dependency relations, filters obsolete nodes, and selects a connected subgraph under a token budget. The retrieved subgraph is serialized in dependency order with provenance and status annotations. The Agent therefore receives the collaboration state required for its decision instead of a fixed window or the full transcript.

When the environment, a tool result, or the task plan changes, CoReMem creates a new version and marks the replaced node as superseded. A dependency engine marks reachable downstream nodes as stale. The recovery scheduler then maps affected node types to operations: reread an updated memory, rerun a tool or analysis, replan a subgoal, or rewrite an artifact. Only after verification or regeneration does an affected node become current, invalid, or superseded. Recovery proceeds in dependency order and leaves nodes outside the affected subgraph frozen. Context-version checks also refresh an Agent before it commits output derived from an older state. All graph writing, retrieval, propagation, and recovery operations remain external to the frozen base LLMs.

Our contributions are:

- We develop a shared heterogeneous memory graph for long-horizon multi-agent execution that unifies distributed collaboration records while retaining ownership, provenance, verification, version, and validity information.
- We propose a task-conditioned local subgraph retriever and a dependency-driven update mechanism that together support selective reading and localized rereading, recomputation, replanning, and artifact rewriting without fine-tuning the underlying LLMs.
- We design an executable evaluation protocol with controlled history length, cross-Agent dependency depth, and delayed state changes. Four research questions test end-to-end consistency, retrieval quality, recovery correctness and cost, and long-horizon scaling.

2 PRELIMINARIES

2.1 LONG-HORIZON MULTI-AGENT EXECUTION

Let q denote a long-horizon task and $\mathcal{A} = \{a_1, \ldots, a_m\}$ a team of language-model Agents. The task is decomposed into subgoals $\mathcal{G}_q = \{g_1, \ldots, g_n\}$. At step t, Agent a_i receives a local context c_t^i, calls a frozen language model π_i^{LLM}, and may emit a message, invoke a tool, update a plan, or

write an artifact. The execution history is an event stream $\tau_{1:t} = (e_1, \ldots, e_t)$. Conventional systems expose this stream through a full transcript, an Agent-private history, a summary, or a vector index.

We distinguish the complete event history from the *collaboration state*: the subset of current facts, intermediate results, decisions, constraints, and artifacts that can affect future execution. The collaboration state is distributed because different Agents produce and consume different portions of it. A memory system must therefore preserve both content and cross-role relations.

2.2 HETEROGENEOUS COLLABORATIVE MEMORY GRAPH

At step t, CoReMem represents memory as a directed typed graph

$$\mathcal{G}_t = (\mathcal{V}_t, \mathcal{E}_t, \psi_V, \psi_E), \tag{1}$$

where ψ_V and ψ_E map nodes and edges to types. The node set is

$$\mathcal{V}_t = \mathcal{V}_A \cup \mathcal{V}_Q \cup \mathcal{V}_G \cup \mathcal{V}_M \cup \mathcal{V}_O \cup \mathcal{V}_S \cup \mathcal{V}_D, \tag{2}$$

corresponding to Agents, tasks, subgoals, messages, tool outputs, environment states, and deliverables. Edge types are

$$\psi_E(e) \in \{\textsc{Delegates}, \textsc{References}, \textsc{DependsOn}, \textsc{ConflictsWith}, \textsc{Precedes}, \textsc{Produces}, \textsc{Consumes}, \textsc{Supersedes}\}. \tag{3}$$

Plans are represented by task and subgoal nodes; user instructions, summaries, and textual analyses are message nodes; and executable return values are tool-output nodes. The distinction preserves the fixed seven-type schema while allowing each record to retain a finer-grained subtype. The five core relations are delegation, reference, dependency, conflict, and time; PRODUCES, CONSUMES, and SUPERSEDES support their execution and version lifecycle. Edge direction follows information or control flow. DELEGATES points from an assigning Agent or goal to its assignee; REFERENCES points from the referenced source to the citing node; and $(u, v) \in \mathcal{E}_{\text{dep}}$ means that v depends operationally on u. PRECEDES points from the earlier event to the later event, SUPERSEDES from an old version to its replacement, and CONFLICTSWITH is stored as two directed arcs. Consequently, retrieval traverses DEPENDSON backward from a candidate decision to its prerequisites, whereas invalidation traverses the same edges forward from a changed prerequisite to its dependents.

Each persistent object has a stable identity $o(v)$ and one or more immutable version nodes. Each version node v stores content x_v, producer $a(v)$, timestamp $t(v)$, version $z(v)$, verification state $\nu(v)$, validity state $\eta(v)$, and validity epoch $b(v)$. Exactly one version per live object may be current. We use

$$\nu(v) \in \{\textsc{Unverified}, \textsc{Verified}, \textsc{Rejected}\}, \tag{4}$$

$$\eta(v) \in \{\textsc{Current}, \textsc{Stale}, \textsc{Invalid}, \textsc{Superseded}\}. \tag{5}$$

Verification records whether available evidence supports the node; validity records whether it can be used in the current execution. A previously verified conclusion can become stale after one of its inputs changes.

2.3 TASK-CONDITIONED LOCAL MEMORY

Before Agent a_i works on subgoal g_j, the system forms a memory query

$$u_t^i = (a_i, g_j, q, p_t, o_t), \tag{6}$$

where p_t is the active plan and o_t the latest observation. A retriever returns an induced subgraph $\mathcal{G}_t^i \subseteq \mathcal{G}_t$ whose serialized representation fits token budget B:

$$\mathcal{G}_t^i = \text{Retrieve}(\mathcal{G}_t, u_t^i; B), \text{Tok}(\text{Serialize}(\mathcal{G}_t^i)) \leq B. \tag{7}$$

The desired subgraph contains not only semantically related nodes but also the dependency paths, provenance, status, and task constraints required for the current decision. Invalid, superseded, and unresolved stale nodes are excluded as current evidence. A compact record may still expose them as a warning or a target that must be revalidated.

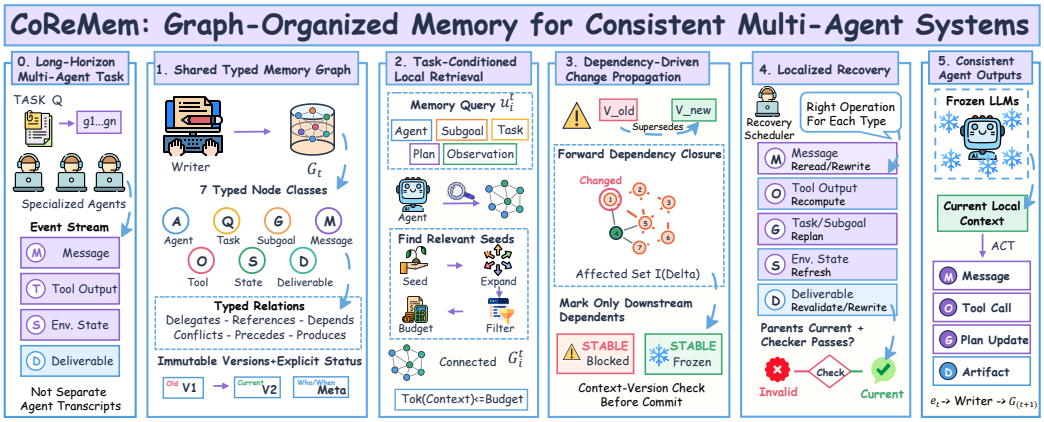

Figure 1: Overview of CoReMem: shared graph memory, task-conditioned retrieval, dependency-aware updates, and localized recovery.

2.4 State Change and Localized Recovery

A state change $\Delta_t = (v, x_v, x'_v)$ replaces, corrects, or withdraws node v. The graph creates a new version v' and records $(v, v') \in \mathcal{E}_{\text{SUPERSEDES}}$, from the old version to its replacement. Let $\mathcal{E}_{\text{dep}}$ contain the operational dependency edges used for propagation. The affected set is the fixed point

$$\mathcal{I}^{(0)}(\Delta_t) = \{v\}, \mathcal{I}(\Delta_t) = \bigcup_{k \geq 0} \mathcal{I}^{(k)}(\Delta_t), \tag{8}$$

$$\mathcal{I}^{(k+1)}(\Delta_t) = \mathcal{I}^{(k)}(\Delta_t) \cup \{w : \exists u \in \mathcal{I}^{(k)}, (u, w) \in \mathcal{E}_{\text{dep}}\}. \tag{9}$$

The replaced node becomes superseded or invalid, while its unresolved descendants become stale until revalidated or regenerated. The recovery procedure selects operations from

$$\mathcal{O} = \{\text{REREAD}, \text{REVALIDATE}, \text{RECOMPUTE}, \text{REPLAN}, \text{REWRITE}, \text{REFRESH}\} \tag{10}$$

and applies them only to $\mathcal{I}(\Delta_t)$ in dependency order. The design goal is to preserve post-change task correctness while minimizing retrieved tokens and recovery operations. CoReMem therefore uses deterministic dependency propagation and type-directed local recovery rather than learning a policy over repair actions.

3 Methodology

CoReMem maintains a shared collaboration state for a team of Agents executing task q over a long event stream. At step t, the active Agent a_i receives query $u_t^i = (a_i, g_j, q, p_t, o_t)$, which specifies its subgoal, current plan, and latest observation. Rather than reconstructing state from private transcripts, it reads a task-conditioned view of the shared graph and writes its resulting event back to the same graph. When a previously consumed record changes, the system revisits its recorded dependents instead of restarting the entire collaboration. This lifecycle has two pillars: graph-organized collaboration memory and dependency-driven memory update. Both operate outside the Agent LLMs and leave their weights unchanged.

3.1 Graph-Organized Collaboration Memory

Shared graph and versioned writing. The typed graph $\mathcal{G}_t = (\mathcal{V}_t, \mathcal{E}_t, \psi_V, \psi_E)$ follows the schema in Section 2: Agents, tasks, subgoals, messages, tool outputs, environment states, and deliverables are nodes; delegation, reference, dependency, conflict, and temporal relations connect them. These relations distinguish the source of a claim from its consumer, and a task assignment from an evidential dependency. The writer maps each execution event to the appropriate node type and records the relations exposed by assignments, citations, tool inputs, file reads, and artifact production. Each persistent object retains a stable identity, while updates create immutable version nodes linked by SUPERSEDES. Agents consuming the same event therefore refer to one shared version, rather than maintaining separate authoritative copies.

Writing preserves read set $\mathcal{R}_t^i$, containing versions consumed to produce event e_t. Let $W(e_t)$ denote its new version nodes, $\mathcal{L}(e_t)$ its typed and version links, and $\mathcal{R}_t(w)$ the logged inputs consumed by output w. Typed triples (u, r, w) allow multiple relations between endpoints; $\mathcal{E}_{\text{dep}}$ projects dependency triples onto endpoint pairs. The update adds event records and dependencies while preserving existing content:

$$\mathcal{V}_{t+1} = \mathcal{V}_t \cup W(e_t), \mathcal{E}_{t+1} = \mathcal{E}_t \cup \mathcal{L}(e_t) \cup \{(v, \text{dep}, w) : w \in W(e_t),\ v \in \mathcal{R}_t(w)\}. \tag{11}$$

The final set in Eq. 11 contains input-to-consumer DEPENDSON edges, typed by ψ_E; a reference alone does not establish dependence. Content is immutable, but validity metadata changes when versions are replaced or rejected. Producer, timestamp, version, verification, and validity metadata distinguish current records from outdated or unresolved ones. Logged dependencies provide an operational trace, not complete knowledge of implicit semantic support.

Task-conditioned local retrieval. Semantic matching and task metadata provide candidates $\mathcal{C}_t^i = \text{Seed}(u_t^i, \mathcal{G}_t)$. The retriever expands them along selected typed relations to depth H, following dependencies backward for prerequisites and other relations for assignments, provenance, constraints, and ordering. This recovers structural context omitted by similarity alone without returning the full dependency closure.

Let $\mathcal{V}_t^{\text{cur}}$ contain versions eligible as current evidence, and let $\ell(S)$ measure tokens in the formatted induced subgraph. Selection balances task-relevant support against context budget:

$$\max_{S \subseteq \mathcal{A}_t^i} \mathcal{U}(S; u_t^i) \quad \text{s.t.} \quad \ell(S) \le B, \mathcal{A}_t^i = \text{Expand}_H(\mathcal{C}_t^i) \cap \mathcal{V}_t^{\text{cur}}. \tag{12}$$

Here $\mathcal{U}(S; u_t^i)$ denotes the task-relevance criterion of the existing selector, evaluated on the records and typed relations in $\mathcal{G}_t[S]$. It expresses the desired relevance–budget trade-off, rather than introducing another learned scorer or claiming an exact combinatorial solver. The budgeted selector returns a feasible local set S_t^i under this objective. Invalid, superseded, and unresolved stale versions are excluded as current evidence. The formatter orders the selected records by dependency and time and includes their types, producers, versions, validity states, and direct provenance. Consequently, the Agent receives a compact view of the collaboration state rather than unrelated text chunks or an unfiltered transcript.

Execution and read-version validation. The frozen Agent LLM acts on u_t^i and context $c_t^i = \text{Fmt}(\mathcal{G}_t[S_t^i])$, producing a message, tool invocation, plan update, or artifact event. Its consumed-version read set persists throughout execution. Since another Agent or observation may change an input before commitment, CoReMem rechecks that set at the write boundary. With c denoting commit time:

$$y_t^i \sim \pi_i^{\text{LLM}}(\cdot \mid u_t^i, c_t^i), \mathcal{G}_c^+ = \begin{cases} \text{Write}(\mathcal{G}_c, e_t; \mathcal{R}_t^i), & \mathcal{R}_t^i \subseteq \mathcal{V}_c^{\text{cur}}, \\ \mathcal{G}_c, & \text{otherwise.} \end{cases} \tag{13}$$

The event e_t packages the execution output. At commit time, Write applies Eq. 11 and validity updates only if every consumed version remains current. Otherwise, the Agent rereads the relevant view and revisits the output. The unchanged branch rejects this candidate write without undoing concurrent events in $\mathcal{G}_c$. Successful commitment closes the read–execute–write loop and exposes provenance to subsequent Agents.

3.2 DEPENDENCY-DRIVEN MEMORY UPDATE

Change-induced invalidation. Revision $\Delta_t = (v, x_v, x_v')$ corrects, replaces, or withdraws version v. The writer preserves historical provenance, stores any replacement, and marks v superseded or invalid. The dependency engine follows input-to-consumer edges to find records requiring reconsideration. Index the $n_t = |\mathcal{V}_t|$ versions with $A_t[u, w] = 1$ exactly when $(u, w) \in \mathcal{E}_{\text{dep}}$. For changed-version indicator $\mathbf{d}_t$, Boolean reachability gives:

$$\mathbf{b}_t = \bigvee_{k=0}^{n_t - 1} (A_t^\top)^k \mathbf{d}_t, \mathcal{I}_t = \{w \in \mathcal{V}_t : \mathbf{b}_t[w] = 1\}. \tag{14}$$

All matrix products and powers in Eq. 14 use the Boolean semiring: multiplication is conjunction and addition is disjunction. The $k = 0$ term includes the changed version, and the remaining terms

include its reachable dependents. This is an equivalent characterization of the least dependency-closed set containing v; the implementation uses graph traversal, not dense matrix powers. Unresolved descendants become stale and cannot be used as current evidence. A downstream record can be affected even when its text is dissimilar to the changed input, because propagation follows recorded consumption rather than lexical similarity. Conversely, temporal adjacency or a reference edge alone does not trigger dependency propagation. Omitted dependencies can still cause incomplete invalidation.

Type-directed local recovery. CoReMem processes affected records in prerequisite-first order whenever their inputs are available. The record type determines the recovery operation: affected messages are reread or rewritten; tool outputs are recomputed with current inputs; task and subgoal nodes trigger local replanning; environment records are refreshed; and deliverables are revalidated or rewritten. Replanning revisits only dependent subgoals and assignments, rather than replacing the entire plan. These are deterministic type-directed operations, not actions selected by an additional learned repair policy.

The frontier starts at $\mathcal{I}_t$ and extends through dependents if repaired outputs introduce further changes. Let $\mathcal{I}_t^\star$ denote the resulting affected versions and $\Omega_t^\star = \{o(v) : v \in \mathcal{I}_t^\star\}$ their objects. Object state $\sigma_c(o)$ stores live version, content, and validity; $\vartheta(o)$ specifies type; and $\mathcal{P}_c(o)$ resolves prerequisites against the current graph. Recovery preserves unaffected states:

$$\sigma_c^+(o) = \begin{cases} \mathrm{Repair}_{\vartheta(o)}\big(\sigma_c(o), \mathcal{P}_c(o)\big), & o \in \Omega_t^\star, \\ \sigma_c(o), & o \notin \Omega_t^\star. \end{cases} \tag{15}$$

The repair operator includes the verification gate below and leaves records stale when prerequisites remain unresolved. Stable object records remain unchanged, while repaired objects may acquire new provenance links. Changed outputs receive immutable versions with current inputs. Successful revalidation can retain content without recomputation; if prerequisite references change, the writer still creates a candidate version with updated provenance rather than reactivating one supported by superseded inputs.

Verification and recovery completion. Revisiting a record does not establish its validity. For candidate $\hat{v}$, let $\mathrm{Pa}_c(\hat{v})$ contain recorded prerequisite versions at validation time c, and let $K_c(\hat{v})$ denote its verification check. Acceptance requires both current prerequisites and successful verification:

$$P_c(\hat{v}) = \bigwedge_{u \in \mathrm{Pa}_c(\hat{v})} [\,\eta_c(u) = \mathrm{Current}\,], \eta^+(\hat{v}) = \begin{cases} \mathrm{Current}, & P_c(\hat{v}) \wedge K_c(\hat{v}), \\ \mathrm{Invalid}, & P_c(\hat{v}) \wedge \neg K_c(\hat{v}), \\ \mathrm{Stale}, & \neg P_c(\hat{v}). \end{cases} \tag{16}$$

A replacement becomes current only after this gate succeeds; its predecessor remains superseded. Unresolved prerequisites leave a candidate stale, while a failed check marks it invalid. The commit-time read-version gate in Eq. 13 also applies during recovery.

Recovery completes when each goal-relevant affected object has a current or invalid latest counterpart, or is explicitly withdrawn as invalid. Historical superseded versions need not become current, but a goal-relevant stale candidate prevents completion. Recovery completion does not imply task success: invalid required deliverables can block the goal. Locality saves replay when changes affect small dependency regions; benefits depend on dependency coverage and impact size. Near-global changes can eliminate savings and leave graph-maintenance overhead.

4 Experiments

We study end-to-end quality and consistency (**RQ1**), task-conditioned retrieval (**RQ2**), delayed-change recovery (**RQ3**), and scaling and failure boundaries (**RQ4**).

4.1 Experimental Setup

Tasks. We use GSM8K, SVAMP, AQuA, and MATH (Cobbe et al., 2021; Patel et al., 2021; Ling et al., 2017; Hendrycks et al., 2021b); HumanEval and MBPP (Chen et al., 2021; Austin et al., 2021); and MMLU, HotpotQA, and BrowseComp+ (Hendrycks et al., 2021a; Yang et al., 2018;

Table 1: Overall results on nine benchmarks. Scores are percentages and higher is better. The final column reports the macro-average change from FH-Recency; this separates effect size from the task scores instead of repeating a delta beneath every entry. Green cells mark the best and second-best values.

	Reasoning				Code		Knowledge/Web				
Method	GSM8K	SVAMP	AQuA	MATH	HumanEval	MBPP	MMLU	HotpotQA	BrowseComp+	Avg.	ΔFH
FH-Recency	87.31	84.72	54.86	60.24	72.56	76.18	77.03	30.92	18.47	62.48	–
ALH	86.44	85.36	55.71	61.83	73.29	77.46	76.58	31.74	20.63	63.23	↑ 0.75
SVM-Chroma	88.19	86.51	56.28	62.77	74.61	78.33	78.11	33.86	23.91	64.73	↑ 2.25
A-MEM	88.73	86.94	57.38	63.91	75.34	78.51	78.84	35.72	25.96	65.70	↑ 3.22
G-Memory	89.08	87.64	57.06	64.29	76.02	79.11	79.36	36.41	27.58	66.28	↑ 3.80
LightMem	89.16	87.38	57.82	64.73	75.91	79.02	79.48	37.69	28.42	66.62	↑ 4.14
MAGMA	89.77	87.12	58.31	65.08	75.68	78.76	80.27	38.47	29.84	67.03	↑ 4.55
Graphiti/Zep	89.24	87.51	58.12	64.96	75.82	78.94	80.35	39.53	32.68	67.46	↑ 4.98
MRAgent	89.62	88.05	58.76	65.81	76.48	79.57	**80.41**	40.86	31.72	67.92	↑ 5.44
CoReMem	**90.26**	**88.73**	**59.67**	**67.34**	**77.15**	**80.02**	80.16	**42.63**	**34.91**	**68.99**	↑ 6.51

Chen et al., 2025). Fixed-role executions expose plans, messages, tool outputs, verification, and artifacts as memory events. We report pass@1 for code, accuracy otherwise, and their unweighted mean.

Methods and controls. RQ1 compares FH-Recency, ALH, SVM-Chroma, A-MEM, G-Memory (Zhang et al., 2025), LightMem, MAGMA (Jiang et al., 2026), Graphiti/Zep, MRAgent, and CoReMem. RQ2 varies only the reader policy on fixed CoReMem graphs; RQ3 varies recovery policy on matched traces, graphs, readers, changes, and injection times.

Metrics and default stress setting. State Consistency is agreement on current object versions; Stale Error counts decisions using invalidated evidence. We also measure retrieval precision/recall/F1, MSR, decision accuracy, affected-set precision/recall, post-change success, operations, and cost/FR. Defaults are four Agents, 128 nodes, depth four, one delayed change affecting about 20%, and a Medium context budget. Backbone, workflow, instances, and budgets are shared. With aggregate results only, all comparisons and fits are descriptive.

4.2 RQ1: End-to-End Collaboration Quality

Table 1 shows that CoReMem achieves the highest observed macro-average (68.99%), exceeding MRAgent by 1.07 points and improving on eight of nine benchmarks. Gains are largest on HotpotQA (+1.77) and BrowseComp+ (+2.23), consistent with benefits on tasks requiring cross-step evidence integration, although these aggregate comparisons do not establish the underlying cause. MRAgent remains marginally stronger on MMLU (+0.25). Figure 2 visualizes the consistency–cost tradeoff: CoReMem achieves 96.11% State Consistency using 25.49K input tokens, compared with MAGMA's 91.73% and 27.64K. Bubble areas encode token consumption, placing CoReMem toward the high-consistency, low-cost region. Relative to MAGMA, it also reduces Stale Error by 63.6%, LLM calls by 9.0%, and tool calls by 13.7%. SVM-Chroma uses fewer tokens (24.32K), but trails CoReMem by 4.26 points in macro-average performance and 12.39 points in State Consistency. Thus, CoReMem does not minimize token usage outright; it achieves a stronger observed balance between task quality, shared-state consistency, and execution cost.

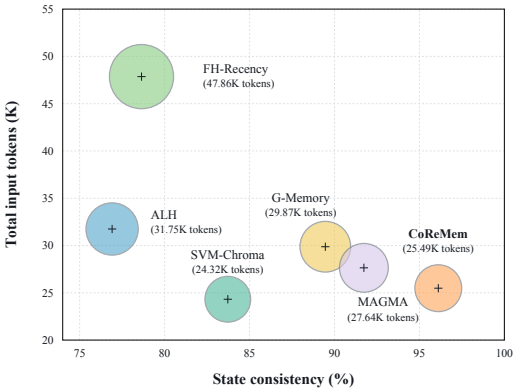

Figure 2: Shared-State Consistency and Token Efficiency in Long-Horizon Multi-Agent Collaboration.

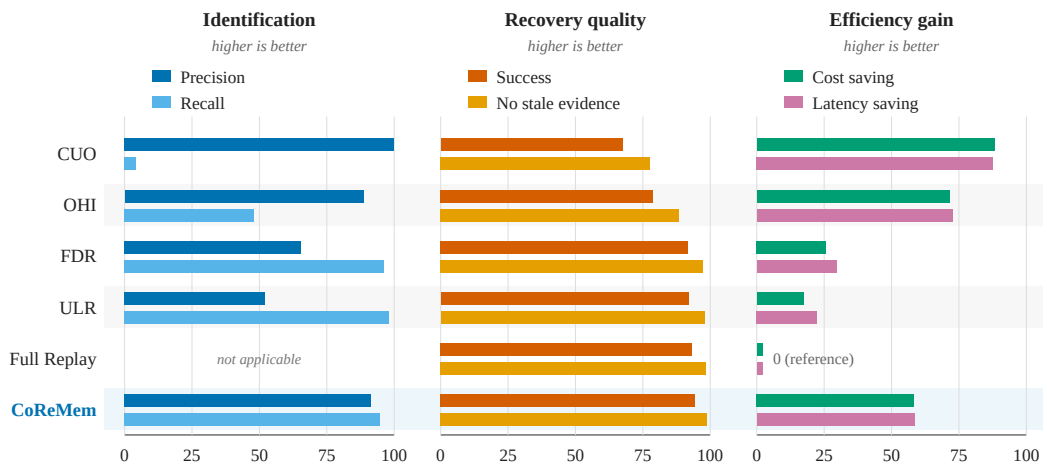

Figure 3: Dependency-Driven Local Recovery Balances Affected-Memory Identification, Post-Change Recovery Quality, and Recovery Efficiency under Delayed Changes.

4.3 RQ2: Task-Conditioned Local Subgraph Retrieval

Table 2: Retrieval quality, downstream decision support, and context efficiency of different memory readers (RQ2). Green cells denote the best and second-best non-reference values; Oracle-MinSupport is shown only as a diagnostic reference.

	Requirement Retrieval			Decision Support			Efficiency	
Reader	Recall ↑	Prec. ↑	F1 ↑	MSR ↑	Dec. Acc. ↑	Inv. Leak. ↓	Nodes ↓	Tokens ↓
Full-Closure	**94.86**	50.38	65.81	**96.92**	82.73	11.84	18.72	3954.61
FixedHop-k	82.61	64.27	72.29	85.38	84.96	0.00	15.64	3328.47
SemanticTopK	78.94	74.62	76.72	81.73	83.88	0.00	**11.83**	**2486.19**
CoReMem-Read	90.73	**81.26**	**85.73**	94.91	**88.21**	0.00	13.92	2967.54
Oracle-MinSupport	72.84	100.00	84.29	100.00	90.47	0.00	8.74	1814.63

MSR denotes minimum-support recall; Dec. Acc. denotes downstream decision accuracy; Inv. Leak. measures retrieval of invalidated memory. Nodes and Tokens report the average retrieved nodes and input tokens, respectively. Oracle-MinSupport is a diagnostic reference and is excluded from best/second-best ranking.

Table 2 shows that CoReMem-Read consistently improves the quality of the context supplied to the downstream Agent on identical frozen memory graphs. Compared with the primary FixedHop-k baseline, it improves Required-Node F1 by 13.44 points, MSR by 9.53 points, and decision accuracy by 3.25 points, while using 10.8% fewer input tokens. Full-Closure achieves higher raw coverage, but indiscriminately expanding the dependency closure reduces precision to 50.38% and introduces 11.84% invalid-memory leakage. CoReMem-Read instead uses 25.0% fewer tokens while improving decision accuracy by 5.48 points, indicating that broader retrieval is not necessarily more useful for the current decision. The comparison with SemanticTopK further isolates the role of graph structure: CoReMem-Read gains 13.18 MSR and 4.33 decision points, suggesting that semantic similarity alone misses structurally necessary support nodes. Finally, CoReMem-Read remains only 2.26 decision points below the non-deployable Oracle-MinSupport upper bound. Because all methods operate on the same frozen graph, candidate pool, budget, serialization, and downstream Agent, these differences specifically support task-conditioned local graph reading rather than differences in graph construction, while leaving cross-system transfer of the reader outside the scope of RQ2.

4.4 RQ3: Dependency-Driven Local Recovery

Figure 3 shows a clear trade-off between propagation completeness and recovery cost. CUO and OHI are inexpensive but recover only 4.18% and 47.93% of affected nodes, leaving downstream state unresolved. Conversely, FDR and ULR exceed 96% affected recall but consume 74.31% and 82.63% of Full Replay (FR) cost. CoReMem-Recover achieves a better balance, with 91.52% precision, 94.76% recall, 94.28% post-change success, and only 1.19% Stale Error at 41.73% of FR

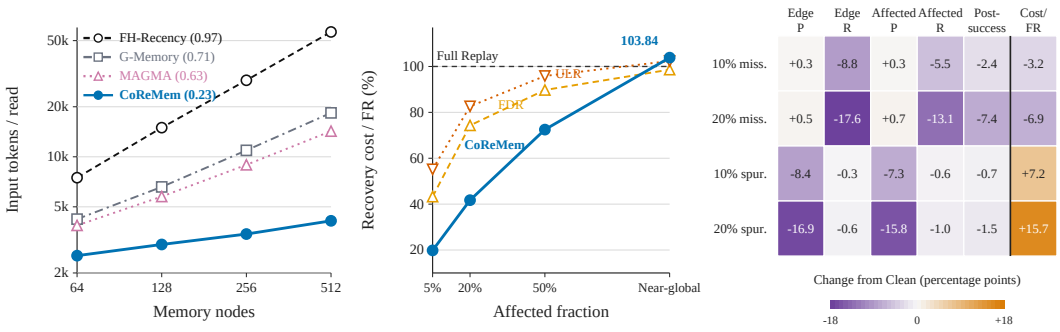

Figure 4: Scaling behavior and failure boundaries of CoReMem under increasing memory size, affected fraction, and dependency noise.

cost. It reduces recovery cost by 43.8% versus FDR and 49.5% versus ULR while retaining comparable coverage. CoReMem also requires only 9.13 rereads, 6.54 recomputations, and 2.38 replans, compared with 29.73, 23.48, and 7.16 under FR. These savings do not result from skipped recovery, as CoReMem achieves the highest post-change success and lowest Stale Error. Overall, the results support selective propagation with type-directed localized recovery over both under-propagation and conservative replay. Since RQ3 varies the recovery strategy jointly, however, the gains support the complete typed-recovery policy rather than its individual components.

4.5 RQ4: Scaling Trends and Failure Boundaries

Figure 4 shows that CoReMem scales substantially more slowly than global-history alternatives as memory grows. From 64 to 512 nodes, FH input increases from 7,486 to 56,392 tokens per read, whereas CoReMem grows only from 2,542 to 4,118, with task success decreasing by 1.48 points. The four-point log–log slopes are 0.969 for FH, 0.710 for G-Memory, 0.629 for MAGMA, and 0.230 for CoReMem ($R^2 \geq 0.997$), consistent with reading a local support subgraph rather than the growing global history. This advantage diminishes as changes become less local: at affected fractions of 5%, 20%, and 50%, recovery costs 19.84%, 41.73%, and 72.48% of Full Replay (FR), reaching 103.84% under near-global change. Treating the latter as $\rho = 1$ yields the descriptive fit $C_{\mathrm{CoReMem}}/C_{\mathrm{FR}} \approx 1.044\rho^{0.557}$ ($R^2 = 0.999$), while missed affected mass $q = \rho(1 - R_{\mathrm{aff}})$ tracks post-change failure as $1 - S_{\mathrm{post}} \approx 0.045 + 1.157q$ ($R^2 = 0.9996$); these fits describe only the tested regime. Dependency noise further exposes distinct failure modes: removing 20% of dependencies reduces affected recall by 13.12 and success by 7.36 points, whereas adding 20% spurious edges reduces affected precision by 15.84 points and raises recovery cost by 15.73 points, with success declining only 1.47 points. Thus, missing dependencies primarily threaten correctness, while spurious dependencies mainly erode efficiency.

5 Conclusion

We introduced CoReMem, a model-agnostic memory framework that organizes long-horizon multi-agent collaboration as a versioned, typed graph. CoReMem retrieves task-conditioned local support for each decision and, when upstream information changes, propagates invalidation through recorded dependencies to repair only the affected subgraph without updating the underlying LLM weights. Across nine benchmarks, CoReMem achieves a 68.99% macro-average, improving by 1.07 percentage points over the strongest graph-memory baseline, while attaining 96.11% State Consistency and reducing Stale Error by 63.6%. Controlled comparisons further identify local reading and dependency-driven recovery as plausible contributors: task-conditioned retrieval improves F1 by 13.44 points over fixed-hop reading, and localized recovery preserves 94.76% affected-node recall at only 41.73% of Full Replay cost. Nevertheless, these aggregate differences remain descriptive rather than causal. Our boundary analyses also expose important limitations: missing dependency edges reduce recovery correctness, spurious edges trigger unnecessary repairs, and localized recovery becomes more expensive than Full Replay in the tested near-global-change setting. Future work should improve dependency capture and uncertainty-aware recovery, while better characterizing the local-to-global crossover. Extending CoReMem to asynchronous agents, evolving environments, and larger heterogeneous teams could further establish its robustness and scalability.

Reproducibility Statement

We describe CoReMem and its evaluation in the main text and appendix. Our evaluation uses nine publicly available benchmarks: GSM8K, SVAMP, AQuA, MATH, HumanEval, MBPP, MMLU, HotpotQA, and BrowseComp+. Code and configurations of CoReMem will be released to facilitate reproduction and support transparent experimental verification.

AI USE STATEMENT

Generative AI tools were not used in the preparation of this manuscript or in the development of the method. CoReMem is evaluated within large language model-based multi-agent systems and therefore uses generative models as experimental backbones and transient executors. These models remain frozen, while memory organization, retrieval, dependency propagation, and recovery are handled externally by CoReMem. This use is an integral part of the evaluated methodology rather than assistance in manuscript writing or technical development.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program synthesis with large language models, 2021.

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, 2021.

Zijian Chen, Xueguang Ma, Shengyao Zhuang, Ping Nie, Kai Zou, Andrew Liu, Joshua Green, Kshama Patel, Ruoxi Meng, Mingyi Su, Sahel Sharifymoghaddam, Yanxi Li, Haoran Hong, Xinyu Shi, Xuye Liu, Nandan Thakur, Crystina Zhang, Luyu Gao, Wenhu Chen, and Jimmy Lin. BrowseComp-Plus: A more fair and transparent evaluation benchmark of deep-research agent, 2025.

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, 2021.

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*, 2021a.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the MATH dataset. In *Advances in Neural Information Processing Systems*, 2021b.

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 *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VtmBAGCN7o.

Shuo Ji, Yibo Li, and Bryan Hooi. Memory is reconstructed, not retrieved: Graph memory for llm agents, 2026.

Dongming Jiang, Yi Li, Guanpeng Li, and Bingzhe Li. Magma: A multi-graph based agentic memory architecture for ai agents, 2026.

Wang Ling, Dani Yogatama, Chris Dyer, and Phil Blunsom. Program induction by rationale generation: Learning to solve and explain algebraic word problems. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, pp. 158–167. Association for Computational Linguistics, 2017. doi: 10.18653/v1/P17-1015.

Yanki Margalit, Nurit Cohen-Inger, Erni Avram, Ran Taig, and Oded Margalit. Governed shared memory for multi-agent llm systems, 2026.

Arkil Patel, Satwik Bhattamishra, and Navin Goyal. Are NLP models really able to solve simple math word problems? In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 2080–2094. Association for Computational Linguistics, 2021. doi: 10.18653/v1/2021.naacl-main.168.

Yiqi Wang, Zihao Yan, Jiaqi Zhang, Zhangkai Wu, Mingkai Zheng, Zequn Sun, Yanming Zhu, and Taotao Cai. Map-graph: Provenance-aware shared memory for multi-agent workflows, 2026.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversations. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=BAakY1hNKS.

Zhaofen Wu, Hanrong Zhang, Fulin Lin, Wujiang Xu, Xinran Xu, Yankai Chen, Henry Peng Zou, Shaowen Chen, Weizhi Zhang, Xue Liu, Philip S. Yu, and Hongwei Wang. Gam: Hierarchical graph-based agentic memory for llm agents, 2026.

Buqiang Xu, Yijun Chen, Jizhan Fang, Ruobin Zhong, Yunzhi Yao, Yuqi Zhu, Lun Du, and Shumin Deng. Structmem: Structured memory for long-horizon behavior in llms, 2026.

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

Guibin Zhang, Muxin Fu, Guancheng Wan, Miao Yu, Kun Wang, and Shuicheng Yan. G-memory: Tracing hierarchical memory for multi-agent systems, 2025.

A Supplementary Evaluation Details

This supplement expands the operational description and empirical analysis of CoReMem. It first clarifies the evaluation scope, then describes the two memory pillars at the level of records and execution transitions, and finally presents auxiliary results, boundary analyses, and an illustrative lifecycle. The supplementary tables provide measurements compressed in the main discussion, including shared-state consistency, operation counts, replanning outcomes, and dependency perturbations.

A.1 Evaluation scope and experimental controls

The four research questions intervene at different levels. RQ1 compares complete memory systems under the shared execution protocol. RQ2 holds the CoReMem graph fixed and substitutes the reading strategy. RQ3 holds the graph, reader, trace, and injected change fixed while substituting the recovery policy. RQ4 changes the operating conditions to examine history growth, impact size, and imperfect dependency records. This separation distinguishes end-to-end system comparisons from controlled interventions on reading and recovery.

The default controlled stress setting uses four Agents, 128 memory nodes, graph dependency depth four, and one delayed change affecting approximately 20% of the graph. Graph dependency depth describes the execution structure and is distinct from the reader's expansion depth H. These quantities define the controlled memory setting rather than intrinsic properties of the nine benchmarks. A memory node represents a typed collaboration record rather than an LLM turn or problem instance; one Agent turn may create or consume multiple versioned records.

RQ1 compares FH-Recency, Agent-local history (ALH), shared vector memory backed by Chroma (SVM-Chroma), A-MEM, G-Memory, LightMem, MAGMA, Graphiti/Zep, MRAgent, and CoReMem under the same execution protocol. FH-Recency provides a budget-constrained transcript reference, ALH tests private history organization, and SVM-Chroma provides a shared non-graph

Table 3: Scope of the reported comparisons and controlled components.

RQ	Changed component	Shared comparison context	Main interpretation
RQ1	Complete memory system	Backbone, workflow, task instances, and budgets	End-to-end quality and resource trade-off
RQ2	Reading policy	Stored graph and downstream decision task	Support selection on the same memory state
RQ3	Update/recovery policy	Reader, trace, graph, change, and injection time	Recovery quality and operational expense
RQ4	History, impact, or edge noise	Corresponding baseline operating protocol	Finite-range robustness and failure boundaries

memory reference. The remaining published memory systems provide structured or graph-based comparisons. RQ1 varies the memory system while keeping the backbone, workflow, task instances, and execution budgets shared across methods.

Dataset and baseline references are provided in the main *Experiments—Experimental Setup* subsection. The operational specifications below expand the *Preliminaries* and *Methodology* sections. RQ1 evaluates complete memory systems, whereas RQ2 and RQ3 provide controlled comparisons of complete reading and recovery policies, respectively.

A.2 Benchmark coverage and interpretation

The evaluation covers four reasoning benchmarks (GSM8K, SVAMP, AQuA, and MATH), two code benchmarks (HumanEval and MBPP), and three knowledge or information-seeking benchmarks (MMLU, HotpotQA, and BrowseComp+). Code tasks use pass@1 and the remaining tasks use accuracy. The reported macro-average assigns equal weight to all nine benchmarks.

The fixed-role workflow represents planning, messages, evidence, tool outputs, checks, and deliverables as memory events, allowing the same memory interface to operate across task families while preserving each benchmark's final answer criterion. History-growth and impact-size interventions separately evaluate behavior as the stored collaboration history and affected region increase.

Memory supervision and answer evaluation measure complementary properties. Required-support annotations identify the information needed by a decision, whereas task checkers evaluate the resulting answer or artifact. Retrieval quality is therefore reported alongside downstream task outcomes rather than used as a substitute for them.

A.3 Reporting conventions and empirical sources

RQ1 reports task-level performance together with shared-state and resource measurements; RQ2 reports retrieval and downstream decision outcomes; RQ3 reports affected-set localization, recovery outcomes, operations, and normalized recovery cost; and RQ4 reports history-growth, impact-size, and dependency-perturbation results. Arithmetic comparisons in the supplement are computed from matched operating conditions using the same metric definitions as the corresponding empirical tables.

The supplementary operational descriptions specify the graph records, reading procedure, version transitions, dependency propagation, and recovery semantics used throughout the evaluation. The worked lifecycle example later in the supplement illustrates these operations on a concrete revision scenario.

B Graph-Organized Collaboration Memory: Operational Details

CoReMem separates an object's identity from the particular version on which a decision relies. This distinction is the link between normal reading and change recovery. An Agent should be able to refer to the same shared object as another Agent while still recording exactly which historical content it consumed. A shared mutable text field cannot provide both properties: overwriting it erases the earlier premise, while copying it into private histories creates multiple competing authorities.

Table 4: Source identifiers for the empirical tables and derived comparisons.

ID	Workbook	Sheet	Cell range
R1T	`rq1.xlsx`	`task`	`A1:K11`
R1S	`rq1.xlsx`	`Sheet2`	`A1:F7`
R2	`rq2.xlsx`	`Sheet1`	`A1:I6`
R3	`rq3.xlsx`	`Sheet1`	`A1:K7`
R4H	`rq4.xlsx`	`read`	`A1:G5`
R4I	`rq4.xlsx`	`recovery`	`A1:G5`
R4N	`rq4.xlsx`	`noise`	`A1:G6`

B.1 Typed records, provenance, and versions

The graph has seven node categories. Agent nodes identify producers and assignees. Task and subgoal nodes represent the active objective, plan structure, and local assignments. Message nodes include instructions, summaries, intermediate claims, and reflection. Tool-output nodes preserve executable return values. Environment-state nodes identify observations of the external state. Deliverable nodes represent outputs subject to final acceptance. These categories are semantic roles, not seven separate memory stores; they share one version-aware graph.

Table 5: Interpretation of the fixed CoReMem record schema. Task and subgoal are distinct node types grouped in one row for exposition; there are seven types in total. Examples do not introduce additional categories.

Node category	Representative content	Collaboration question answered
Agent	Producer or assigned role	Who produced or owns this work?
Task / subgoal	Objective, plan, local constraint	Which objective does this record serve?
Message	Instruction, claim, summary, reflection	What was asserted or communicated?
Tool output	Returned evidence or execution result	What did the tool actually return?
Environment state	Observed file, data, or external condition	Which external state was assumed?
Deliverable	Answer, program, or report	Which output must remain acceptable?

A record's stable identity persists across revisions. Its content version is immutable, while validity and verification metadata can change. Replacing an observation therefore creates a successor version and a SUPERSEDES link rather than editing the earlier content. The predecessor remains available for tracing earlier decisions, but it is no longer eligible as current evidence. At most one version of a live object may be current; withdrawal or unresolved recovery can leave an object with no current version. The phrase "shared state" thus means a common authoritative version history, not that every stored object is always usable.

The graph also separates verification from temporal validity. Verification records whether an available check supports a candidate; validity records whether that candidate is eligible in the present execution. A correctly verified analysis can become stale when its source is revised. Conversely, a newly written record is not established as correct merely because it is the newest. Keeping these fields separate prevents both mistakes: treating old verification as permanent authorization, and treating recency as proof.

Typed edges carry complementary information. Delegation records control ownership; references record citation; dependencies record operational consumption; conflict identifies incompatible claims; and temporal edges preserve ordering. Production, consumption, and supersession links support the event and version lifecycle. Only the recorded operational dependencies determine the invalidation closure. A temporally adjacent message can help retrieval reconstruct an exchange without being a premise that must be recomputed after the earlier message changes.

B.2 Task-conditioned reading and context assembly

The reading query combines the active Agent, subgoal, overall task, current plan, and latest observation. Semantic matching proposes content candidates, while task metadata narrows them to the current collaboration context. Typed expansion supplies prerequisites, assignments, provenance, and ordering information that may not be lexically similar to the query. For dependency edges directed

from an input to its consumer, prerequisite retrieval follows the edges backward. Invalidation later follows the same edges forward; confusing these directions would exchange an answer's evidence with the outputs that depend on it.

The candidate graph is filtered for current evidence and selected under the serialized token budget. The utility expression in the main method specifies a relevance–budget objective; it does not introduce a trained scoring network or establish that an exact combinatorial optimizer is implemented. The selected graph need not be a complete transitive closure. If a limited budget cannot accommodate all relevant evidence, the appropriate empirical questions are how much required support survives, how much irrelevant content remains, and whether the downstream decision stays correct. RQ2 measures these outcomes directly.

Serialization should preserve the relationships that made the selected records useful. A tool result without its producer, input version, and validity state can be mistaken for a current observation even when its text is accurate for an older environment. The method therefore includes type, producer, version, validity, and direct provenance alongside the selected content, ordered by dependency and time. These fields are part of the context cost: comparing raw payload length alone would understate the price of an informative graph view.

Stale and superseded versions can be exposed as warnings or repair targets without becoming accepted premises. For example, a context may say that an earlier analysis requires revalidation, while withholding that analysis from the current evidence set. This distinction matters for interpreting invalid-memory leakage. Simply counting every mention of an old version as an error would penalize an explicit warning; counting it as valid because it appears in the prompt would permit outdated evidence to reenter reasoning. The relevant distinction is its intended evidential use.

B.3 READ–EXECUTE–WRITE SPECIFICATION

The operational sequence below expands the existing first pillar. It introduces no additional learned component. Its read manifest records the versions actually consumed by an output where that information is available, rather than treating every visible node as a proven semantic dependency.

Procedure: local memory transaction.

[leftmargin=*,itemsep=2pt,topsep=3pt]

1. Form the task-conditioned query and obtain seed records from semantic and task metadata.
2. Expand the permitted typed relations, retain eligible current evidence, and select a feasible context within the serialized token budget.
3. Execute the frozen Agent on the query and formatted local graph. Preserve the consumed-version read manifest with the candidate event.
4. At commitment, resolve the read manifest against the current graph. If any consumed version is no longer current, reject this candidate write and revisit the relevant decision using a refreshed view.
5. Otherwise, add immutable event versions, typed provenance, and input-to-consumer dependency links. Expose the accepted event through the same shared graph.

The commit-time check addresses a specific race: an Agent may read a current version and finish its reasoning only after another Agent has replaced that version. Checking only at retrieval time cannot detect this situation. Rejecting the candidate write leaves concurrent accepted events intact; it does not reset the graph to the earlier snapshot. The manifest consequently acts as a dependency record and as a freshness check at the publication boundary.

This is a memory-commit condition, not a general transactional guarantee for external tools. A tool may already have changed a file or another external system before the candidate memory write is rejected. Idempotence, sandboxing, compensation, and user authorization remain properties of the tool-execution layer. Similarly, metadata checks can establish only the freshness of recorded inputs; they cannot detect an omitted private premise. These qualifications delimit what the first pillar is designed to maintain and motivate the dependency-noise analysis.

C Dependency-Driven Memory Update: Operational Details

A revision changes the admissibility of downstream information even when that information's text remains unchanged. CoReMem responds by separating impact detection from repair. Detection identifies records whose support must be reconsidered; repair determines the type of work required to produce acceptable current counterparts. Marking a record stale is therefore not a finding that its conclusion is false, and revisiting a record is not sufficient to make it current again.

C.1 Invalidation frontier and traversal

Propagation starts from the replaced or withdrawn version and follows input-to-consumer dependency edges. A queue-based traversal implements the Boolean reachability characterization in the main method: initialize the queue with changed versions, visit each discovered dependency successor, and stop when no new successor remains. A visited set prevents a cycle from causing unbounded traversal. The result is the least set closed under the recorded dependencies, not an oracle set of all semantic consequences.

The changed historical version becomes superseded when replaced, or invalid when withdrawn. Its unresolved dependents become stale. Distinguishing these states avoids suggesting that an obsolete source itself should eventually become current. Recovery is instead concerned with the current counterpart of each affected object. Historical content and lineage remain in the graph so that an earlier decision can be explained, but those old nodes do not have to be reactivated for the recovery episode to finish.

Propagation deliberately excludes bare temporal and citation links. If a verifier cites an unrelated earlier exchange only to explain its workflow, that citation does not necessarily make the verification depend on the exchange's content. Conversely, a numerical report can depend on a data record with no shared keywords. Typed operational links capture the latter connection without automatically converting the former into mandatory replay. This selectivity is conditional on the dependency records being accurate; absent links cause missed propagation, and spurious links broaden the frontier.

For one traversal, work is proportional to the dependency vertices and edges visited, assuming ordinary adjacency access and constant-time membership checks. This observation concerns the traversal primitive only. It is not a bound on total recovery time, which also includes queries, serialization, verification, tool execution, replanning, and potentially repeated attempts under further changes. A small closure may contain an expensive tool call, while a larger closure may contain inexpensive records whose content can be retained after checking.

C.2 Type-directed actions and prerequisites

The second pillar uses deterministic record-type rules. It does not learn a repair policy from rewards or choose actions using a probabilistic truth-maintenance model. Table 6 gives the interpretation of those rules. An action label describes the semantic repair operation; it does not imply that all actions of that type have identical runtime or token cost.

Table 6: Type-directed repair interpretation. Every successful candidate still requires current prerequisites and a successful applicable check.

Affected record	Relevant action	What must be reconsidered
Message or summary	Reread / rewrite	Claims or instructions formed from changed premises
Tool output	Recompute	Tool invocation using the current input versions
Task or subgoal	Replan	Dependent local objective, assignment, or constraint
Environment state	Refresh	Observation of the relevant external state
Deliverable	Revalidate / rewrite	Acceptance against current evidence and requirements

Recovery proceeds prerequisite first when the needed inputs are available. It begins with the detected frontier and extends that frontier when repaired outputs introduce further dependent changes. Local

preservation applies to the resulting affected object set: objects outside it retain their content and validity state. New provenance may connect repaired records to stable inputs, so this preservation should not be misread as forbidding every new incident edge on an otherwise unchanged object.

Revalidation can retain a conclusion when its content remains supported, but retaining content is different from retaining obsolete lineage. If a source version has changed, an accepted candidate must point to the current prerequisite version. The writer therefore creates a new candidate with updated provenance even when the text happens to remain identical. Reactivating the old version would conceal which premise was checked and could leave another Agent reading a historically valid but currently unsupported record.

Local replanning follows the same principle. A revised requirement may invalidate one subgoal or assignment without requiring a new plan for the entire task. The corresponding subgoal is revisited, its necessary prerequisites are resolved, and its dependents are updated if its result changes. RQ3 reports replanning separately because it can be operationally important even though a plan is represented using the existing task/subgoal types rather than a new plan-node category.

C.3 Verification, stopping conditions, and unresolved states

For each repaired candidate, the validity gate asks two questions: are all recorded prerequisites current, and does the applicable verification check pass? A candidate becomes current only when both conditions hold. It becomes invalid when current prerequisites are available but verification fails. Otherwise it remains stale. The check is task-dependent; the aggregate results do not reveal the exact implementation of every checker, so no stronger guarantee about checker soundness is inferred here.

Procedure: dependency-local recovery.

1. Record the revision, retain historical lineage, and traverse recorded dependencies from the changed version.
2. Mark unresolved affected dependents stale. Resolve affected versions to their persistent objects and initialize the repair frontier.
3. Select an affected object whose prerequisites can be resolved. Apply its type-directed operation using current inputs; leave it unresolved if those prerequisites are unavailable. An explicitly withdrawn object can instead be recorded invalid; withdrawal does not automatically invalidate every descendant.
4. Check the candidate's prerequisites and verification result. Publish an accepted current version or an explicit invalid outcome; reject stale-input candidate writes at the same commit boundary used during normal execution.
5. Extend the frontier if the accepted repair affects additional dependents. Continue until every goal-relevant affected object has a current or invalid latest counterpart, or report unresolved recovery.

The invalidation traversal terminates on a finite graph with a visited set, but successful recovery need not follow. Cyclic unresolved prerequisites may leave no eligible record, a tool can fail repeatedly, or a new revision can arrive during repair. The method's completion rule does not justify silently accepting such states. They remain unresolved unless the available evidence or task structure permits progress. No particular cycle-breaking policy or retry threshold is claimed by the current aggregate evaluation.

Recovery completion must also be separated from final task success. Explicitly marking a withdrawn source and its unsupported deliverable invalid can finish state reconciliation while leaving the task unsolved. This outcome is preferable to calling the stale deliverable current, but it must still count as a task failure under an answer or artifact checker. Reporting post-change success alongside affected-set quality and resource usage preserves this distinction.

D Metric Definitions and Resource Accounting

This section defines the metrics and resource-accounting conventions used throughout the experiments. Set-based measures are evaluated at the corresponding decision or revision level, while benchmark-level task scores follow the evaluation protocol described in Section 4.

D.1 TASK QUALITY, SHARED STATE, AND MEMORY SUPPORT

The macro task score is the arithmetic mean of the nine benchmark scores. Differences between score percentages are reported in percentage points (pp). State Consistency measures whether relevant Agents agree on the current version of a shared object rather than merely producing similar textual responses. Agreement on an obsolete version does not count as consistent. Stale Error measures decisions that use non-current information as present evidence; historical versions retained for provenance are not themselves counted as stale errors.

For a decision, let S be the selected memory-node set, R the annotated required-node set, and $M^\star$ the designated minimal-support set. Required-node precision and recall measure retrieval with respect to R, whereas minimal-support recall (MSR) measures coverage of $M^\star$:

$$P_{\text{req}} = \frac{|S \cap R|}{|S|}, \qquad R_{\text{req}} = \frac{|S \cap R|}{|R|}, \qquad \text{MSR} = \frac{|S \cap M^\star|}{|M^\star|}. \tag{17}$$

The broader required set R and the sufficient minimal-support set $M^\star$ serve different purposes: the former evaluates overall support retrieval, whereas the latter tests whether the essential support needed for the decision is present. Oracle-MinSupport targets $M^\star$ directly and therefore need not retrieve every node in R.

F1 summarizes the balance between required-node precision and recall. Decision accuracy measures the correctness of the downstream response produced from the retrieved context. Invalid-memory leakage measures exposure of non-current information as accepted evidence in that context. These metrics capture complementary failure modes: a reader may avoid stale evidence while missing required support, or retrieve sufficient current support while the downstream Agent still produces an incorrect decision.

D.2 AFFECTED-SET AND RECOVERY OUTCOMES

For each injected revision, we construct the reference affected set $I^\star$ independently of the recovery policy being evaluated. Starting from the controlled revision, an offline reference pass traces the recorded execution and checks downstream records using their task-level inputs, deterministic tool or program recomputation when available, and the corresponding answer, unit-test, or evidence checker. A record is included in $I^\star$ if the revision requires its content, validity, or downstream output to be reconsidered under this reference pass. CoReMem's predicted propagation set, stale labels, and recovery decisions are not used to construct $I^\star$.

Let $\widehat{I}$ denote the affected set predicted by the evaluated recovery policy. We measure

$$P_{\text{aff}} = \frac{|\widehat{I} \cap I^\star|}{|\widehat{I}|}, \qquad R_{\text{aff}} = \frac{|\widehat{I} \cap I^\star|}{|I^\star|}. \tag{18}$$

Affected-set precision measures unnecessary propagation, whereas recall measures affected records missed by the recovery policy. These localization metrics are distinct from repair correctness. Post-change success measures whether the recovered task succeeds, while post-change Stale Error measures continued reliance on obsolete evidence. Full Replay does not predict a localized affected set and therefore has no affected-set precision or recall.

Reread, recompute, and replan count distinct classes of recovery operations. Replan success measures outcomes on replanned subgoals, whereas post-change success measures the overall recovered task. Operation counts are reported separately because a single affected object may require more than one recovery operation.

D.3 COSTS AND ARITHMETIC COMPARISONS

Resource measurements retain their experimental accounting scope. RQ1 reports execution-level input tokens, LLM calls, and tool calls; RQ2 reports retrieved context tokens for each reading decision; and the RQ4 history sweep reports input tokens per read. These quantities therefore measure different stages of the memory lifecycle and are compared only within their corresponding experimental setting.

Recovery cost is normalized to Full Replay within each matched comparison, with Full Replay assigned a reference value of 100. For a lower-is-better quantity x, the relative reduction of CoReMem against baseline b is

$$\text{Reduction}_b(x) = 100\left(1 - \frac{x_{\text{CoReMem}}}{x_b}\right)\%. \tag{19}$$

Positive values indicate lower CoReMem usage or cost, whereas negative values indicate higher usage relative to the specified baseline. Percentage-point differences are used instead for percentage-valued quality metrics such as task success, consistency, precision, and recall.

All resource and quality comparisons are made between matched conditions within the same experiment. Recovery-cost values are reported relative to Full Replay, while latency comparisons use the common latency measure within each corresponding recovery setting.

E Additional End-to-End Results

E.1 Shared-state quality and complete resource profile

Table 7 reports the full auxiliary RQ1 results. CoReMem has the highest reported State Consistency at 96.11% and the lowest Stale Error at 2.07%. The corresponding MAGMA values are 91.73% and 5.69%. Their difference is 4.38 consistency points and a 63.62% relative reduction in stale error. These quantities describe related but distinct benefits: agreement on shared current versions and avoidance of invalidated evidence during decisions.

Table 7: Complete shared-state and resource results from RQ1. Consistency and stale error are percentages; the remaining entries are reported aggregate counts. (Table 4).

Method	Consistency ↑	Stale error ↓	Input tokens ↓	LLM calls ↓	Tool calls ↓
FH-Recency	78.64	14.73	47,862.41	49.38	16.24
ALH	76.91	16.28	31,748.62	45.76	14.87
SVM-Chroma	83.72	10.96	24,318.57	42.91	14.22
G-Memory	89.46	7.38	29,872.33	40.68	13.09
MAGMA	91.73	5.69	27,641.85	39.42	12.67
CoReMem	**96.11**	**2.07**	**25,487.26**	**35.86**	**10.93**

Table 8: Derived CoReMem changes relative to each reference. All reductions are percentages of the reference value; negative token reduction denotes higher CoReMem usage. Consistency gains are percentage points (pp). Source R1S (Table 4).

Reference	Cons. gain (pp)	Stale reduction	Token reduction	LLM reduction	Tool reduction
FH-Recency	17.47	85.95	46.75	27.38	32.70
ALH	19.20	87.29	19.72	21.63	26.50
SVM-Chroma	12.39	81.11	-4.81	16.43	23.14
G-Memory	6.65	71.95	14.68	11.85	16.50
MAGMA	4.38	63.62	7.79	9.03	13.73

The complete resource profile qualifies the quality advantage. CoReMem uses 25,487.26 input tokens, 35.86 LLM calls, and 10.93 tool calls in the reported aggregate. Against MAGMA, the reductions are 7.79%, 9.03%, and 13.73%, respectively. Against FH-Recency, the corresponding reductions are 46.75%, 27.38%, and 32.70%. The gain is therefore not confined to prompt compression: the aggregate call counts are also lower. These measurements do not reveal which individual calls were avoided, so they should not be interpreted as a direct count of redundant decisions removed by one specific mechanism.

SVM-Chroma remains the lowest-token system at 24,318.57 input tokens. CoReMem uses 4.81% more tokens when SVM-Chroma is the denominator, while improving consistency by 12.39 points and the reported macro task score by 4.26 points. Equivalently, SVM-Chroma is approximately 4.6% cheaper when CoReMem is the denominator. The full comparison therefore supports a quality–resource trade-off, not a claim that CoReMem minimizes all forms of computation.

ALH further illustrates why fewer tokens are not sufficient evidence of a better memory system. Its token usage is below FH-Recency, but its State Consistency is lower and its Stale Error is higher. Private histories can be economical while leaving relevant Agents with incompatible current views. This observation is compatible with the motivation for shared memory, although a system-level table alone cannot attribute the difference exclusively to duplication or state drift.

Table 8 reports every pairwise comparison under one denominator convention. Positive reductions mean less CoReMem usage; the negative token entry against SVM-Chroma preserves the exception. CoReMem improves all reported shared-state metrics and call counts, while retaining a modest token premium over the most economical vector baseline.

E.2 Task-level variation and the macro-average

Table 9: Task-level differences derived from the main-result workbook. The best baseline is selected separately within each benchmark, not by macro-average. Source R1T (Table 4).

Benchmark	CoReMem	MAGMA	Δ MAGMA (pp)	Δ best baseline (pp)
GSM8K	90.26	89.77	0.49	0.49
SVAMP	88.73	87.12	1.61	1.09
AQuA	59.67	58.31	1.36	1.36
MATH	67.34	65.08	2.26	2.26
HumanEval	77.15	75.68	1.47	1.13
MBPP	80.02	78.76	1.26	0.91
MMLU	80.16	80.27	-0.11	-0.11
HotpotQA	42.63	38.47	4.16	4.16
BrowseComp+	34.91	29.84	5.07	5.07

The main table reports a 68.99% CoReMem macro-average and 67.03% for MAGMA, a 1.96-point difference at the displayed precision. Table 9 shows how this difference is distributed. CoReMem improves over MAGMA on eight benchmarks and trails it on MMLU by 0.11 points. The largest differences are 4.16 points on HotpotQA and 5.07 points on BrowseComp+. The smaller positive differences on reasoning and code tasks show that the overall gain is not uniform across task families.

The strongest macro-average baseline need not be the strongest baseline on every benchmark. G-Memory exceeds MAGMA on SVAMP, HumanEval, and MBPP. CoReMem's gaps against the per-task best baseline on these datasets are therefore 1.09, 1.13, and 0.91 points, respectively, rather than the larger gaps against MAGMA. Reporting both comparisons prevents an overall baseline ranking from overstating every task-specific improvement.

The relatively large information-seeking gains are consistent with the hypothesis that keeping cross-step evidence current is useful. They do not establish that dependency length causes the differences. The task table contains neither measured dependency lengths nor a matched manipulation of them. Similarly, the small MMLU deficit is not sufficient to conclude that graph memory is harmful for knowledge questions. Both interpretations would exceed the resolution of the available evidence.

The macro-average is a concise summary rather than a complete characterization. It gives a one-point change on BrowseComp+ the same arithmetic contribution as a one-point change on GSM8K, despite their different absolute score levels. We retain this weighting because it is the declared metric, and accompany it with per-task differences and shared-state measurements. We do not replace it with a post hoc task weighting selected to favor CoReMem.

F Additional Retrieval Results

F.1 Reader variants and controlled scope

The RQ2 comparison evaluates alternative ways of reading the same stored collaboration graph. Full-Closure returns the available dependency closure without the task-conditioned selection used by CoReMem. FixedHop-k uses a fixed-depth neighborhood. SemanticTopK selects using semantic relevance without dependency expansion. CoReMem-Read combines task conditioning, typed expansion, budgeted selection, and current-evidence filtering. Oracle-MinSupport receives designated true support and serves only as a privileged diagnostic.

Table 10: Complete retrieval-quality results, in percentages. Oracle-MinSupport is a privileged diagnostic. Required-node and minimal-support metrics have different reference sets. Source R2 (Table 4).

Reader	Req. recall	Req. precision	Req. F1	MSR	Decision acc.
Full-Closure	94.86	50.38	65.81	96.92	82.73
FixedHop-k	82.61	64.27	72.29	85.38	84.96
SemanticTopK	78.94	74.62	76.72	81.73	83.88
CoReMem-Read	**90.73**	**81.26**	**85.73**	**94.91**	**88.21**
Oracle-MinSupport	72.84	100.00	84.29	100.00	90.47

Table 11: Retrieval resources and invalid-memory exposure from RQ2. The workbook reports aggregate node counts and token usage; zero leakage is an observed aggregate, not a universal guarantee. Source R2 (Table 4).

Reader	Invalid leakage (%)	Retrieved nodes	Input tokens
Full-Closure	11.84	18.72	3,954.61
FixedHop-k	0.00	15.64	3,328.47
SemanticTopK	0.00	11.83	2,486.19
CoReMem-Read	**0.00**	**13.92**	**2,967.54**
Oracle-MinSupport	0.00	8.74	1,814.63

Holding the stored graph fixed removes graph construction as the changing factor within RQ2. It does not make every pairwise comparison a one-component ablation. In particular, Full-Closure reports 11.84% invalid leakage, whereas the other readers report zero. The Full-Closure comparison therefore changes evidence coverage and exposure to invalid records together. It evaluates the complete reading policy under that setup; it does not isolate the effect of task conditioning independently of validity filtering.

F.2 Coverage, precision, and decision quality

CoReMem-Read obtains 90.73% required-node recall, 81.26% precision, and 85.73% F1. Relative to FixedHop-k, its reported F1 improves by 13.44 points and decision accuracy by 3.25 points. Input tokens fall from 3,328.47 to 2,967.54, a 10.84% reduction. The joint direction is consistent with a more selective context that retains relevant structural support, rather than obtaining accuracy merely by returning a larger neighborhood.

Full-Closure has higher required recall and MSR, at 94.86% and 96.92%, but its precision is 50.38% and decision accuracy is 82.73%. CoReMem-Read sacrifices 4.13 required-recall points and 2.01 MSR points while improving precision by 30.88 points and decision accuracy by 5.48 points. It also uses 24.96% fewer tokens. These results exhibit a coverage–selectivity trade-off: the largest returned dependency region is not necessarily the best decision context in the reported setting.

SemanticTopK is more economical than CoReMem-Read, using 2,486.19 tokens and 11.83 retrieved nodes. CoReMem-Read uses 2,967.54 tokens and 13.92 nodes, while improving MSR from 81.73% to 94.91% and decision accuracy from 83.88% to 88.21%. Thus the comparison does not show a uniformly smaller context. It suggests that some additional structurally relevant evidence is worth retaining, subject to the observed increase in reading cost.

The zero leakage entries should be interpreted at their reported aggregate precision. They are consistent with current-evidence filtering in this evaluation, but they do not establish that every future graph or concurrent execution is free of stale input. A missing dependency can leave an outdated conclusion incorrectly marked current, which a status filter alone cannot recognize. This is why RQ4 perturbs the dependency records rather than treating filtering as a complete solution to validity maintenance.

F.3 What the oracle diagnostic does and does not establish

Oracle-MinSupport reports 100% precision and 100% MSR, but only 72.84% required-node recall. This combination is not inherently contradictory: the designated minimal-support set can be smaller

than the broader required-node annotation. CoReMem-Read's higher required-node F1 than the oracle is likewise not evidence that it surpasses privileged information on the oracle's own objective. The two metrics reward different coverage targets.

The oracle's decision accuracy is 90.47%, compared with 88.21% for CoReMem-Read. The 2.26-point gap indicates remaining headroom under this particular support annotation and downstream decision procedure. It is not a universal upper bound on Agent performance. Even a sufficient support set does not guarantee that a frozen LLM will interpret it correctly, and a different ordering or representation could affect its response. The oracle is excluded from deployable-system claims and is not described as an ablation, since it adds unavailable information rather than removing a CoReMem component.

G Additional Recovery Results

G.1 Recovery policies and localization behavior

The recovery comparison keeps the graph and reader fixed while replacing the response to a delayed change. Changed-unit-only recovery (CUO) primarily revisits the changed source. One-hop invalidation (OHI) extends recovery to immediate dependents. Full descendant replay (FDR) follows recorded dependencies and uniformly replays the resulting descendants. Untyped lineage replay (ULR) conservatively propagates over untyped lineage relations. Full Replay restarts the complete comparison scope. CoReMem-Recover combines typed dependency propagation with type-directed local operations. ULR is a controlled policy variant rather than an additional published system.

Table 12: Recovery quality in percentages. Full Replay does not predict a localized affected set and therefore has no affected-set precision or recall. Source R3 (Table 4).

Policy	Aff. precision	Aff. recall	Post-change success	Stale error
CUO	100.00	4.18	67.42	22.36
OHI	88.64	47.93	78.86	11.72
FDR	65.37	96.28	91.63	2.73
Full Replay (FR)	–	–	93.16	1.57
ULR	52.18	98.11	91.97	2.14
CoReMem-Recover	**91.52**	**94.76**	**94.28**	**1.19**

Table 13: Recovery operations, replanning outcomes, normalized recovery cost, and latency. Source R3 (Table 4).

Policy	Reread	Recompute	Replan	Replan SR (%)	Cost/FR (%)	Latency
CUO	0.83	0.46	0.18	31.74	11.86	15.63
OHI	4.91	3.27	0.96	63.58	28.47	34.72
FDR	18.47	13.86	4.62	88.47	74.31	89.26
Full Replay (FR)	29.73	23.48	7.16	91.38	100.00	126.84
ULR	22.64	16.93	5.47	86.72	82.63	98.51
CoReMem-Recover	**9.13**	**6.54**	**2.38**	**95.46**	**41.73**	**52.68**

Table 12 separates affected-set localization from end-task recovery. CUO achieves 100% affected precision but only 4.18% recall, identifying only a small fraction of the reference affected set. Its post-change success is 67.42%, with 22.36% Stale Error. OHI increases affected recall to 47.93%, but its 78.86% post-change success and 11.72% Stale Error indicate that one-hop propagation remains insufficient for many delayed changes.

FDR and ULR favor broader propagation. Their affected recall reaches 96.28% and 98.11%, respectively, while precision decreases to 65.37% and 52.18%. Their normalized recovery costs reach 74.31% and 82.63% of Full Replay. These results show that high affected-set recall alone does not imply selective recovery: broad propagation identifies most affected records but also revisits substantially more of the execution.

CoReMem-Recover achieves 91.52% affected precision and 94.76% recall, together with 94.28% post-change success and 1.19% Stale Error. Although its affected recall is slightly below FDR and

ULR, its higher precision limits unnecessary propagation while maintaining strong downstream recovery. The difference between localization and final recovery further motivates evaluating affected-set quality together with post-change task outcomes.

G.2 Operation counts, replanning, and latency

CoReMem-Recover uses 9.13 rereads, 6.54 recomputations, and 2.38 replans, compared with 29.73, 23.48, and 7.16 under Full Replay. These correspond to reductions of 69.29%, 72.15%, and 66.76%, respectively. Its normalized recovery cost is 41.73% of Full Replay, a 58.27% reduction. Relative to FDR and ULR, CoReMem reduces recovery cost by approximately 43.8% and 49.5%, respectively.

CoReMem-Recover also reaches 95.46% replan success, compared with 91.38% for Full Replay, 88.47% for FDR, and 86.72% for ULR. Replan success characterizes the replanned subgoals, whereas post-change success evaluates the recovered task as a whole; the two metrics therefore capture complementary stages of recovery.

Latency decreases from 126.84 under Full Replay to 52.68 under CoReMem-Recover, a 58.47% reduction. CUO and OHI incur lower latency, but their substantially lower post-change success and higher Stale Error illustrate the trade-off between minimum intervention and recovery quality. CoReMem instead occupies an intermediate operating point that combines high localization quality, strong post-change success, and reduced recovery cost.

G.3 Interpretation of the recovery comparison

CoReMem-Recover achieves 94.28% post-change success, compared with 93.16% for Full Replay, while requiring only 41.73% of its normalized recovery cost. This comparison shows that revisiting the complete execution is not necessary to maintain strong recovery performance when the affected region can be localized.

FDR and ULR differ from CoReMem in both propagation scope and repair behavior, so the comparison evaluates each recovery policy as a complete strategy. Under the delayed-change setting, CoReMem combines high affected-set precision and recall with low Stale Error and substantially less replay work. Together with the controlled localization and operation measurements, these results characterize the quality–cost trade-off of the complete typed-recovery policy.

H Scaling Trends and Dependency-Noise Boundaries

RQ4 examines three pressures on the memory lifecycle. History growth increases the amount of stored collaboration context, impact growth expands the region requiring recovery, and dependency perturbations alter the graph used to guide propagation. We evaluate these factors separately because they affect different aspects of memory cost and recovery quality.

H.1 History growth and local reading

Table 14: History-size sweep. Method columns report input tokens per read, and Success reports CoReMem task success. Source R4H (Table 4).

History	FH tokens/read	G-Memory	MAGMA	CoReMem	Success (%)
64 nodes	7,486.32	4,216.47	3,864.19	2,541.73	69.42
128 nodes	14,973.85	6,587.24	5,764.91	2,967.54	68.99
256 nodes	28,864.17	10,932.68	8,947.53	3,428.86	68.71
512 nodes	56,391.74	18,376.42	14,265.81	4,118.37	67.94

Table 14 shows the history sweep from 64 to 512 memory nodes. Over this eightfold increase, FH input grows from 7,486.32 to 56,391.74 tokens per read, a factor of 7.53. G-Memory and MAGMA grow by factors of 4.36 and 3.69, respectively, whereas CoReMem increases from 2,541.73 to 4,118.37 tokens per read, a factor of only 1.62. The widening gap is consistent with CoReMem selecting a local support view rather than repeatedly processing the accumulated history.

Relative to FH, CoReMem reduces input tokens by 66.05% at 64 nodes and 92.70% at 512 nodes. Its task success changes from 69.42% to 67.94%, a decrease of 1.48 points across the sweep. Thus, context consumption grows substantially more slowly than stored history while task performance remains comparatively stable over the evaluated range.

The corresponding four-point log–log slopes are 0.969 for FH, 0.710 for G-Memory, 0.629 for MAGMA, and 0.230 for CoReMem. These slopes summarize the observed scaling behavior over the 64–512-node range and provide a compact comparison of history sensitivity across the four memory strategies.

H.2 Impact size and the limit of locality

Table 15: Impact-size sweep. Recovery costs are normalized to Full Replay; affected recall and post-change success report CoReMem recovery outcomes. Source R4I (Table 4).

Affected fraction	FDR cost	ULR cost	CoReMem cost	Aff. recall	Post-success
5%	43.26	55.18	19.84	93.81	95.06
20%	74.31	82.63	41.73	94.76	94.28
50%	89.74	95.86	72.48	95.13	92.67
Near-global	98.62	102.37	103.84	96.02	90.83

At affected fractions of 5%, 20%, and 50%, CoReMem requires 19.84%, 41.73%, and 72.48% of Full Replay cost, corresponding to savings of 80.16%, 58.27%, and 27.52%. The benefit decreases as the affected region expands because progressively less unaffected work can be preserved. This trend directly reflects the locality principle underlying selective recovery.

The near-global condition exposes the opposite regime. CoReMem reaches 103.84% of Full Replay cost, compared with 98.62% for FDR and 102.37% for ULR. When most goal-relevant execution is affected, preserving stable branches provides little benefit and the graph-based recovery overhead can outweigh the work saved by localization. This endpoint therefore identifies a practical boundary of local recovery rather than an outlier to the general trend.

Affected recall remains high across the sweep, increasing from 93.81% at 5% impact to 96.02% under near-global change, while post-change success decreases from 95.06% to 90.83%. Increasing impact therefore raises recovery difficulty even when the affected region continues to be localized with high recall. Together, the cost and quality trends show that CoReMem is most advantageous when revisions remain substantially local.

H.3 Missing and spurious dependencies

Table 16: Dependency perturbations. All entries are percentages. Clean denotes the recorded graph without added dependency perturbations. Source R4N (Table 4).

Perturbation	Edge P	Edge R	Affected P	Affected R	Post-success	Cost/FR
Clean	95.18	93.47	91.52	94.76	94.28	41.73
10% missing	95.46	84.62	91.84	89.31	91.86	38.54
20% missing	95.71	75.83	92.17	81.64	86.92	34.87
10% spurious	86.74	93.18	84.26	94.21	93.57	48.91
20% spurious	78.31	92.86	75.68	93.74	92.81	57.46

Table 17: Changes relative to the clean operating point. Entries are percentage-point differences, including normalized recovery cost. Source R4N (Table 4).

Perturbation	Δ aff. precision	Δ aff. recall	Δ success	Δ cost/FR
10% missing	0.32	-5.45	-2.42	-3.19
20% missing	0.65	-13.12	-7.36	-6.86
10% spurious	-7.26	-0.55	-0.71	7.18
20% spurious	-15.84	-1.02	-1.47	15.73

The clean graph reaches 95.18% dependency-edge precision and 93.47% recall. Here, "Clean" denotes the baseline recorded graph before additional dependency perturbations. The missing- and

spurious-edge conditions then stress two complementary dependency failures: omitted dependencies restrict propagation, whereas additional dependencies broaden the recovery frontier.

Missing dependencies primarily harm affected-set recall and downstream recovery. With 10% missing edges, affected recall decreases from 94.76% to 89.31%, while post-change success falls from 94.28% to 91.86%. At 20% missing edges, these metrics decrease further to 81.64% and 86.92%, corresponding to drops of 13.12 and 7.36 points from the clean condition. Normalized recovery cost simultaneously decreases to 34.87%, reflecting reduced propagation rather than improved recovery at preserved quality.

Spurious dependencies produce the complementary pattern. At 20% spurious edges, affected precision decreases by 15.84 points to 75.68%, while normalized recovery cost increases by 15.73 points to 57.46%. Affected recall decreases by only 1.02 points and post-change success by 1.47 points. Thus, missing dependencies primarily induce under-propagation and reduce recovery correctness, whereas spurious dependencies induce over-propagation and primarily increase unnecessary recovery work.

The 128-node history setting, 20% impact condition, and clean dependency condition correspond to the default operating points used in the controlled reader and recovery evaluations. RQ4 extends these defaults by varying one operating dimension at a time, allowing history growth, impact size, and dependency quality to be examined separately.

I Illustrative Lifecycle, Limitations, and Further Evaluation

I.1 A worked revision example

We illustrate the CoReMem lifecycle with a team preparing a short report from a tool-provided data table. A retrieval Agent obtains source version $s^{(1)}$; an analysis Agent derives result $a^{(1)}$; and a writing Agent produces deliverable $d^{(1)}$. A separate branch prepares a formatting specification $f^{(1)}$ from unchanged user requirements.

The operational dependencies are $s^{(1)} \rightarrow a^{(1)} \rightarrow d^{(1)}$ and $f^{(1)} \rightarrow d^{(1)}$. These edges indicate operational consumption rather than textual co-occurrence. The writer's local context contains the current analysis and applicable formatting constraint together with their provenance, while unrelated records from other subgoals remain outside the selected view.

Suppose the source tool output is corrected to $s^{(2)}$ after the deliverable has been drafted. The graph retains $s^{(1)}$ as superseded and records the new version. Forward propagation from $s^{(1)}$ reaches $a^{(1)}$ and $d^{(1)}$, marking their unresolved counterparts stale. It does not reach $f^{(1)}$, because the formatting specification does not depend on the corrected source. This directionality allows local recovery to preserve the stable formatting branch while reconsidering its affected consumer.

The analysis is recomputed or revalidated using $s^{(2)}$. If the numerical conclusion changes, a new analysis version records the updated result and current input. If the conclusion remains unchanged, the content can be retained after verification while the accepted counterpart records the updated provenance. In either case, $a^{(1)}$ remains associated with its historical input rather than being reused as current evidence. The deliverable is then checked or rewritten against the accepted analysis and unchanged formatting specification.

Table 18: Illustrative revision lifecycle under dependency-driven local recovery.

Object	Before revision	On source correction	After successful local repair
Source	$s^{(1)}$ current	$s^{(1)}$ superseded	$s^{(2)}$ current
Analysis	$a^{(1)}$ current	Prior result stale	Current counterpart supported by $s^{(2)}$
Formatting	$f^{(1)}$ current	Unchanged	Same $f^{(1)}$ retained
Deliverable	$d^{(1)}$ current	Prior deliverable stale	Current counterpart passes the applicable check

If the revised source makes the requested result unsupported, the deliverable can instead become explicitly invalid. State reconciliation then terminates without treating an unsupported artifact as current. Likewise, if the source changes again while the writer is revising the report, the commit-time read-version gate rejects a candidate derived from the superseded input and triggers reconsideration under the current version.

Dependency errors affect this lifecycle in complementary ways. Omitting the source-to-analysis edge can leave the old analysis outside the propagation frontier even after the source is updated. Adding a spurious source-to-formatting edge instead triggers unnecessary reconsideration of a stable branch. These cases illustrate the under-propagation and over-propagation behaviors examined by the dependency-perturbation experiments.

I.2 Practical limitations and responsible use

CoReMem relies on observable consumption records to construct operational dependencies. Explicit tool inputs, references, assignments, and artifact production provide strong lineage signals, while dependencies arising only from private Agent reasoning are harder to capture. Consequently, recovery quality depends on the coverage and precision of the recorded dependency graph. The missing- and spurious-edge experiments characterize the effects of these two failure modes and motivate improved dependency capture.

Object identity introduces a complementary challenge. Shared identities prevent multiple Agents from maintaining competing authoritative copies only when semantically corresponding events are mapped to the same object. Paraphrased claims, aliases, and partially overlapping evidence may require additional entity resolution or semantic deduplication. Moreover, shared state provides consistency rather than truth: an incorrect observation can remain consistently represented across the team. Verification quality and source reliability therefore remain important alongside memory consistency.

A unified memory graph also requires appropriate access control in practical deployments. Provenance records, tool outputs, and environment states may contain sensitive information, motivating permission-aware retrieval, retention policies, and controls over replayable tool actions. Prompt injection, privacy protection, and irreversible external actions constitute separate security concerns and should be handled jointly with the corresponding tool-execution and authorization mechanisms.

The current evaluation focuses on controlled history growth, delayed changes, impact size, and dependency perturbations. More dynamic settings with larger teams, deeper dependency structures, repeated revisions, and concurrent changes provide natural extensions for evaluating the same memory lifecycle under increasingly complex collaboration patterns.

I.3 Future directions

Three directions follow naturally from the present results. First, factorial reader and recovery experiments can further separate the effects of task conditioning, validity filtering, dependency typing, and type-directed operation selection while holding the remaining execution pipeline fixed.

Second, denser impact-size sweeps can characterize the transition from localized recovery to Full Replay as an increasing fraction of the execution becomes affected. Such measurements can also support an adaptive recovery strategy that selects between localized repair and broader replay according to the estimated impact region.

Third, repeated and concurrent changes provide a natural extension of the versioned recovery protocol. Future evaluations can examine revisions that arrive during recovery, overlapping affected regions, and sequences in which later updates supersede earlier corrections. These settings would test how versioned provenance and dependency-local repair behave under continuously evolving shared state.

Overall, the supplementary results characterize CoReMem across end-to-end task quality, local retrieval, delayed-change recovery, history growth, impact size, and dependency perturbations. Together, they show how a versioned graph can maintain shared collaboration state, provide task-conditioned local views, and use recorded dependencies to localize reconsideration after upstream

changes. The observed boundary cases further identify dependency quality and change locality as central factors governing the effectiveness of selective recovery.