REQMEMBENCH: BENCHMARKING CODING AGENTS FOR TEMPORAL REQUIREMENT-STATE RECONSTRUCTION IN EVOLVING PROJECTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding agents increasingly join ongoing software projects, where they must modify an evolving codebase in the context of a long-running client collaboration. Before acting, an agent must determine which requirements currently govern the task. This requires tracing how requirements have changed over time and determining which still apply, while distinguishing changes to the requirements themselves from evidence about their implementation. Existing coding and memory benchmarks offer limited visibility into whether agents correctly reconstruct the requirement state that should guide implementation. We introduce ReqMemBench, a benchmark for temporal requirement-state reconstruction built from real freelance project histories that have been rewritten for privacy and include a subset of role-played messages. At intermediate takeover points, ReqMemBench provides evidence-linked gold requirement states and evaluates four capabilities: selecting relevant historical requirements and evidence, reconstructing their pre-task states, updating requirements or identifying material clarification needs, and delivering the requested code changes. A two-phase protocol freezes history-based responses before agents choosing to act on eligible tasks receive repository access. Full History and Oracle Relevant History conditions assess the effect of history filtering and agents' reconstruction accuracy when relevant evidence is explicitly supplied. Across seven configurations, agents over-select historical evidence and do not benefit from Oracle Relevant History; even the best RQ3 Post-State Score is only 0.1578, and no system passes one quarter of eligible RQ4 targets. Reliable takeover therefore requires coherent temporal context and stronger update-or-clarify reasoning.

1 INTRODUCTION

Code-generation evaluation has progressively expanded from self-contained function synthesis to repository-scale software engineering. HumanEval and MBPP evaluate short programs generated from natural-language specifications (Chen et al., 2021; Austin et al., 2021). CrossCodeEval and RepoBench introduce cross-file dependencies and repository context (Ding et al., 2023; Liu et al., 2024), while SWE-bench evaluates patches for real GitHub issues and SWE-Lancer evaluates freelance software-engineering tasks using executable or expert-derived criteria (Jimenez et al., 2024; Miserendino et al., 2025). These benchmarks have substantially increased the scope and realism of coding evaluation. However, they generally assume that the task to be implemented is adequately specified by the current issue or prompt. An agent joining an ongoing project faces an additional prerequisite: before modifying the code, it must determine what the project currently requires.

This prerequisite motivates our central research question: *Can a coding agent reconstruct the requirements that currently govern a task from a long and evolving project history, determine whether those requirements are sufficiently resolved to act, and implement the task accordingly?* Answering this question requires more than retrieving relevant messages. In a long-running project, requirements may be introduced, refined, deferred, resumed, or removed. A later update may alter only part of an earlier requirement, while execution feedback may change the known implementation status without changing the requirement itself. Consequently, previously authoritative statements may no longer apply in full, yet may still provide essential evidence about how the current requirement

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

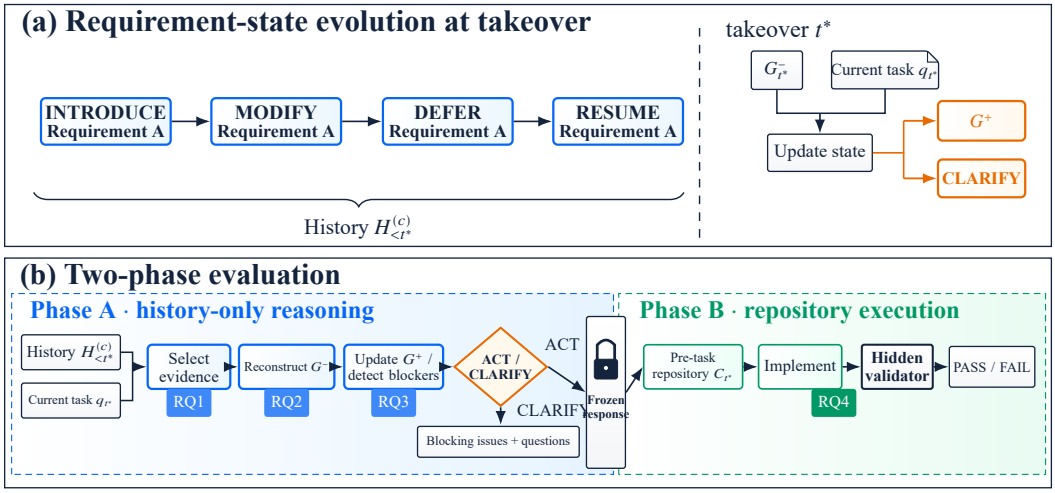


Figure 1: The ReqMemBench takeover setting. An agent first reasons from the preceding history and current task without repository access. After its response is frozen, eligible `ACT` runs receive the pre-task repository for delivery evaluation.

emerged. The agent must integrate these updates to recover the task-relevant requirement state at takeover, including the currently applicable constraints, their scope and lifecycle status, unresolved ambiguities, and available evidence about their implementation.

Several lines of work evaluate capabilities related to this problem. Conversational coding benchmarks examine how models revise programs in response to iterative execution or natural-language feedback (Han et al., 2025). Long-term memory benchmarks evaluate information extraction, temporal reasoning, knowledge updates, long-range understanding, selective forgetting, and abstention over extended interactions (Wu et al., 2025; Maharana et al., 2024; Hu et al., 2025). These benchmarks test whether agents can use historical information, and some additionally evaluate retrieval quality. However, they do not explicitly treat the current task-relevant requirement state as a gold evaluation target. As a result, they cannot separately determine whether an unsuccessful implementation arose because the agent selected the wrong history, reconstructed the requirements incorrectly, failed to recognize a material ambiguity, or implemented a correctly understood requirement poorly.

Figure 1 summarizes the takeover setting and two-phase evaluation. We formulate this problem as *temporal requirement-state reconstruction* and introduce ReqMemBench to evaluate it. Given the interaction history preceding a takeover point and a new request, an agent first reconstructs the pre-task state of each directly affected requirement immediately before the request. It must then incorporate the new request and either produce the complete post-task requirement states or identify ambiguities that materially prevent safe implementation. ReqMemBench derives evidence-linked gold states from annotated, transformed freelance project histories and evaluates four capabilities: history selection, state reconstruction, update-or-clarify reasoning, and code delivery. To assess historical reasoning independently of repository access, the protocol freezes the agent's history-based response before eligible action runs receive the pre-task repository. Comparing Full History with an audited Oracle Relevant History subset shows that relevance filtering does not improve RQ2–RQ4; even the best RQ3 Post-State Score is only 0.1578, revealing a sharp failure to convert recovered evidence into actionable requirement states. Our contributions are:

- We formulate temporal requirement-state reconstruction for coding-agent takeover, with explicit targets for pre-task states, post-task updates, and material clarification needs.
- We develop an auditable annotation-and-construction pipeline that turns privacy-transformed project conversations into evidence-linked Requirement Atoms and Events, deterministically replays them into temporal State Graphs, and materializes takeover instances for RQ1–RQ4. Human review validates semantic annotations, while deterministic checks enforce structural consistency and leakage controls.

- Using this pipeline, we construct ReqMemBench from transformed longitudinal freelance project histories, with task-local gold states and supporting evidence at intermediate takeover points.
- We provide a two-phase evaluation protocol that separately scores history-based requirement reasoning and repository delivery, with paired history conditions for examining the effect of oracle history filtering.

2 Related Work

Coding benchmarks. Coding benchmarks cover function-level synthesis and library use (Chen et al., 2021; Austin et al., 2021; Zhuo et al., 2025), cross-file completion (Ding et al., 2023; Liu et al., 2024), repository issue resolution and freelance engineering (Jimenez et al., 2024; Miserendino et al., 2025), and release-level evolution and project development (Le et al., 2025; Lu et al., 2026). Sequential benchmarks address knowledge transfer, trajectory reuse, and code-quality degradation under evolving specifications (Joshi et al., 2025; Zhu et al., 2026; Orlanski et al., 2026). RigorBench additionally evaluates engineering process discipline, including abstention and clarification (Madiraju & Madiraju, 2026).

Memory and long context. LongBench and RULER evaluate retrieval and reasoning over extended inputs (Bai et al., 2024; Hsieh et al., 2024). LongMemEval, MemoryAgentBench, and LoCoMo assess complementary capabilities across interactions, including temporal and multi-session reasoning, knowledge updates, selective forgetting, and abstention (Wu et al., 2025; Hu et al., 2025; Maharana et al., 2024). These evaluations cover both access to historical information and its integration and revision.

Conversational coding. ConvCodeWorld and RECODE-H evaluate iterative code revision under execution or simulated expert feedback (Han et al., 2025; Miao et al., 2026), while SR-Eval studies stepwise requirement refinement (Zhan et al., 2025). LoCoEval introduces conflicting requirement information in repository-oriented conversations and separately evaluates information-item extraction and function generation at the end of a conversation (Liu et al., 2026). ReqMemBench focuses on task-local requirement states at intermediate takeover points, explicitly tracking attributes, lifecycle, scope, ambiguity, and implementation evidence. Its protocol freezes reconstruction and update-or-clarify responses before repository access and evaluates them alongside eligible code delivery.

3 Problem Formulation

At an arbitrary takeover point, an agent must determine which earlier requirements govern the current request, what state they have reached, and how the request changes them before acting on the codebase. Building on requirements traceability (Nuseibeh & Easterbrook, 2000; Gotel & Finkelstein, 1994), we formulate this problem using time-indexed requirement states and task-local pre- and post-target gold. These objects support four research questions that separate evidence selection, state reconstruction, request interpretation, and repository delivery.

3.1 Temporal State and Task-Local Gold

We represent a project as $P = (H, C)$, where $H = \langle h_1, \ldots, h_n \rangle$ is the ordered interaction history and $C(t)$ is the corresponding code state. A target request q_{t^*} defines a strict temporal boundary: $H_{<t^*}$ and C_{t^*-} contain only pre-target evidence. We use t^{*+} for the point after incorporating requirement changes expressed by the target but before any subsequent implementation or execution feedback.

The atomic unit is a *Requirement Atom*. For atom r, its state is

$$G_r(t) = \big(A_r(t), L_r(t), S_r(t), U_r(t), X_r(t)\big), \tag{1}$$

where A_r records attribute values, L_r the lifecycle, S_r persistence and scope, U_r unresolved ambiguities, and X_r the latest implementation evidence. Unknown fields remain explicit. These dimensions are independent: an active requirement can be ambiguous or have failed implementation evidence, while a removed requirement retains its prior attributes and evidence under lifecycle `REMOVED`.

States are reconstructed by deterministically replaying the validated event sequence E_r up to time t:

$$G_r(t) = \mathrm{Replay}\big(E_r^{\leq t}\big), \tag{2}$$

using the transition rules in Appendix A. Let $R_P(t)$ be the atoms introduced by time t, and let A_{t^*} be all atoms affected by the target, including newly introduced ones. The task-local pre- and post-target gold states are

$$\begin{aligned} R_{t^*}^{\mathrm{hist}} &= A_{t^*} \cap R_P(t^{*-}), \\ G_{t^*}^{-} &= \big(G_r(t^{*-})\big)_{r \in R_{t^*}^{\mathrm{hist}}}, \\ G_{t^*}^{+} &= \big(G_r(t^{*+})\big)_{r \in A_{t^*}}. \end{aligned} \tag{3}$$

Thus, atoms introduced by the target are absent from $G_{t^*}^{-}$ but included in $G_{t^*}^{+}$. Operationally, $G_{t^*}^{-}$ specifies what the agent must recover before interpreting the request, while $G_{t^*}^{+}$ specifies the state that should hold after the request is incorporated. Their difference captures the requirement update induced by the current task. This task-local projection evaluates selection, reconstruction, and updating without requiring recovery of unrelated project state.

3.2 Research Questions and Evaluation Protocol

The four research questions form a diagnostic sequence. RQ1 asks whether the agent can identify the affected historical requirements and their supporting evidence. RQ2 tests whether it can reconstruct the pre-target states of those requirements. RQ3 tests whether it can incorporate the current request into complete post-target states or identify a material ambiguity that requires clarification. For actionable targets, RQ4 tests whether that interpretation can be realized correctly in the repository. Table 1 formalizes the target and access conditions for each question.

Full History (C1) exposes $H_{<t^*}$, whereas Oracle Relevant History (C2) exposes an audited, order-preserving subsequence containing the evidence required to determine the same task-local state and action. C2 isolates reasoning over relevant evidence from the additional problem of finding that evidence in a long history.

Evaluation separates history reasoning from code execution. In Phase A, the agent receives the condition-specific history and current request, but no repository:

$$\big(H_{<t^*}^{(c)}, q_{t^*}\big) \longrightarrow \Big(\widehat{R}_{t^*}^{\mathrm{hist}}, \widehat{G}_{t^*}^{-}, \widehat{d}_{t^*}, \widehat{Z}_{t^*}\Big), \tag{4}$$

where $c \in \{\mathrm{C1}, \mathrm{C2}\}$ and $\widehat{d}_{t^*} \in \{\mathrm{ACT}, \mathrm{CLARIFY}\}$. The output contains selected historical atoms and evidence, their reconstructed pre-task states, and either the complete predicted post-task state $\widehat{G}_{t^*}^{+}$ for `ACT` or material blocking issues and questions $\widehat{Q}_{t^*}$ for `CLARIFY`. Clarification is correct only when visible evidence leaves a target-relevant ambiguity that blocks a determinate update.

The Phase-A response is frozen before repository access. For an execution-eligible target, Phase B begins only after a parseable `ACT` response, at which point the agent receives a fresh copy of $C_{t^{*-}}$. Hidden validators remain inaccessible to the agent, separating requirement-state prediction from repository correctness.

Together, the four questions localize where a takeover attempt fails along the path from project history to code delivery. RQ1 uses only C1 because C2 would reveal its evidence-selection target. Comparing C1 and C2 in RQ2–RQ4 then distinguishes failures caused by finding the relevant evidence from failures in state reconstruction, request interpretation, or execution.

Table 1: Research questions and their corresponding evaluation stages. C1 is Full History and C2 is Oracle Relevant History. RQ4 applies only to eligible targets.

Stage	Evaluated capability	Gold target	History	Repository
RQ1	Select affected historical atoms and supporting evidence	$R_{t^*}^{\text{hist}}$ and supporting evidence groups	C1	No
RQ2	Reconstruct pre-task states	$G_{t^*}^{-}$ restricted to matched atoms	C1/C2	No
RQ3	Update requirements or identify material blockers	$G_{t^*}^{+}$ for `ACT`; blocking issues and questions for `CLARIFY`	C1/C2	No
RQ4	Implement the requested changes	Final repository passing the frozen validator	C1/C2	After freeze

4 REQMEMBENCH CONSTRUCTION

Stage 1 in Figure 2 converts privacy-preserving project timelines into the temporal representation in Section 3. Model-assisted annotation proposes Requirement Atoms, evidence-linked Events, and their source messages; schema checks reject malformed objects, human reviewers correct semantic boundaries and lifecycle changes, and deterministic replay produces an evidence-traceable State Graph. Failed invariants stop construction rather than trigger model-based repair.

In Stage 2, the graph is searched for takeover points that depend on earlier project context and induce a verifiable Requirement transition. A fixed rubric removes context-free requests and implementation-only evidence, while human review adjudicates borderline candidates. Replay immediately before and through each accepted target yields the pre- and post-task snapshots, whose affected-Requirement projection becomes the task-local gold. The same reviewed trajectories produce C1 and the order-preserving C2 subset.

Each frozen target then yields the four evaluation objects: RQ1 uses affected historical Requirements and their supporting evidence; RQ2 uses their pre-task States; RQ3 uses the target-updated States or material blockers that require clarification; and eligible RQ4 instances pair the same pre-task repository with a frozen validator. Agent-visible histories are serialized separately from gold annotations, future messages, repositories, and evaluator artifacts.

Human audit is the semantic authority throughout this pipeline: reviewers validate Event meaning and evidence links, target dependence, ambiguity-sensitive gold, C2 sufficiency, and executable-task validity, while code enforces schema, ordering, replay, and leakage constraints. Disagreements are adjudicated before the benchmark snapshot is frozen. Table 2 reports the frozen corpus and per-RQ instance counts; Appendix B gives the full annotation, selection, isolation, and audit procedures.

Table 2: Frozen construction snapshot. Paired entries report C1/C2 instances.

Stage	Frozen output	Count
Stage 1	Projects; Requirement Atoms; Requirement Events	51; 859; 2,793
Stage 2	Takeover targets (source projects); RQ3 gold branches	349 (44); 312 `ACT`, 37 `CLARIFY`
RQs	RQ1 C1; RQ2 C1/C2; RQ3 C1/C2; RQ4 C1/C2	349; 349/349; 349/349; 37/37

5 EXPERIMENTS

We evaluate seven model–harness configurations under the takeover protocol in Section 3.2, reporting requirement-state reasoning in Phase A and repository delivery in Phase B.

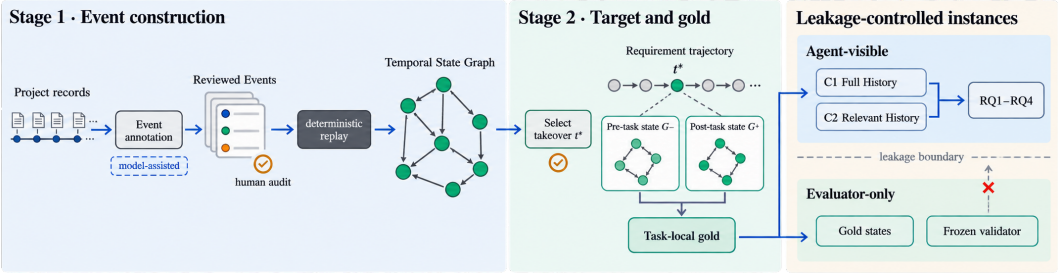


Figure 2: From project records to audited, leakage-controlled ReqMemBench instances.

5.1 Experimental Setup

Systems and controls. Table 3 lists four Codex and three Claude Code configurations. We treat each model–harness pair as one deployed system and compare systems directly, without attributing differences to the harness alone. Within every applicable C1/C2 pair, the current request, prompt, response schema, tools, and resource budgets are identical, and each run starts in a fresh workspace. The experiment manifest freezes the remaining provider, harness, sampling, retry, and stopping settings and retains the Phase-A response, execution repository, and evaluator outputs. Appendix C specifies the full run and failure-handling protocol.

Table 3: Evaluated model–harness configurations. Model identifiers and run settings follow the experiment configurations and operator prompts.

Harness	Model identifier	Reasoning	Harness revision
Codex	`gpt-6-astra`	high	`codex-cli 0.155.1`
Codex	`gpt-5.6-sol`	high	`codex-cli 0.155.1`
Codex	`gpt-5.6-terra`	high	`codex-cli 0.155.1`
Codex	`gpt-5.5`	high	`codex-cli 0.155.1`
Claude Code	`claude-fable-5.1`	high	`claude-code` $\geq$ `2.1.259`
Claude Code	`claude-opus-5.5`	high	`claude-code` $\geq$ `2.1.259`
Claude Code	`claude-sonnet-5`	high	`claude-code` $\geq$ `2.1.259`

Scoring and aggregation. RQ1 reports Requirement and Evidence Precision, Recall, and F1 after one-to-one semantic alignment. RQ2 reports typed Attribute Reconstruction on matched historical Requirements and Reconstruction Coverage over the full target set. RQ3 reports Decision and Balanced Accuracy together with branch-specific Post-State, blocker, and question scores. RQ4 reports repository PASS Rate, requiring build, target-test, and regression success, alongside Executable Coverage. Main-text metrics are macro-averaged over targets; cross-project claims use project macro-averages, and C2–C1 effects use valid paired targets with identical gold. Appendix C defines the comparators, calibration, error handling, and diagnostic metrics.

5.2 Phase A: Requirement-State Reasoning

RQ1: Can agents select the relevant historical requirements? Table 4 shows that Claude Fable 5.1 leads both F1 measures. Selection is recall-heavy across systems: Requirement Recall is at least as high as precision in every row, and Evidence Recall always exceeds Evidence Precision. Even the best Requirement Precision (0.5634) and Evidence Precision (0.6238) remain below their corresponding recalls, while Evidence Recall reaches 0.9564. Systems therefore cover much of the gold evidence by admitting substantial non-gold material rather than by isolating a compact requirement set; the dominant aggregate error is over-selection.

Table 4: RQ1 requirement and evidence selection under Full History (C1). The Targets column gives the frozen RQ1 pool size.

Harness	Model	Targets	Req. P	Req. R	Req. F1	Evid. P	Evid. R	Evid. F1
Codex	GPT-6 Astra	349	0.5178	0.7586	0.6155	0.6037	0.9564	0.7402
Codex	GPT-5.6 Sol	349	0.3943	0.6667	0.4900	0.4656	0.9412	0.5923
Codex	GPT-5.6 Terra	349	0.4286	0.4286	0.4286	0.2984	0.5238	0.3610
Codex	GPT-5.5	349	0.4172	0.5038	0.4564	0.3436	0.6379	0.4466
Claude Code	Claude Fable 5.1	349	0.5634	0.7669	0.6496	0.6238	0.9487	0.7527
Claude Code	Claude Opus 5.5	349	0.5381	0.7337	0.6209	0.5879	0.9126	0.7151
Claude Code	Claude Sonnet 5	349	0.4683	0.6387	0.5404	0.4731	0.8176	0.5994

RQ2: Can agents reconstruct the current requirement state? Table 5 shows that Claude Fable 5.1 leads under both conditions, yet the best coverage remains below 0.79 and the best reconstruction score below 0.57. The two metrics expose distinct failures: limited coverage means that some target requirements are not reconstructed, whereas the reconstruction score shows that even matched requirements retain incorrect or incomplete attributes. Both measures decrease under C2 for every system, so supplying relevant history improves neither target coverage nor state fidelity.

Table 5: RQ2 pre-task state reconstruction. Δ is the paired C2–C1 difference in Attribute Reconstruction. N gives the frozen RQ2 target pool in C1/C2; Reconstruction Coverage retains this full target denominator.

Harness	Model	N C1/C2	Attribute Reconstruction			Coverage	
			C1	C2	Δ	C1	C2
Codex	GPT-6 Astra	349/349	0.5372	0.5068	−0.0304	0.7631	0.7087
Codex	GPT-5.6 Sol	349/349	0.4384	0.3715	−0.0669	0.6667	0.5467
Codex	GPT-5.6 Terra	349/349	0.3667	0.2825	−0.0842	0.4786	0.4444
Codex	GPT-5.5	349/349	0.3971	0.3184	−0.0787	0.5428	0.4726
Claude Code	Claude Fable 5.1	349/349	0.5627	0.5389	−0.0238	0.7816	0.7423
Claude Code	Claude Opus 5.5	349/349	0.5268	0.4973	−0.0295	0.7472	0.7038
Claude Code	Claude Sonnet 5	349/349	0.4578	0.4163	−0.0415	0.6537	0.6079

RQ3: Can agents produce the correct requirement update or clarification? Table 6 shows that Claude Fable 5.1 leads both decision metrics, but every system declines under C2. Because an all-`ACT` policy attains 0.894 raw accuracy on this imbalanced task, Balanced Accuracy is the more informative measure; under C2, three systems fall below its 0.5 all-`ACT` baseline. Even the strongest Balanced Accuracy is 0.7184 in C1 and 0.6632 in C2, leaving substantial error across the two decision classes despite the higher raw accuracy.

Table 6: RQ3 update-or-clarify decision quality. Gold A/C gives the frozen benchmark counts of 312 Gold-ACT and 37 Gold-CLARIFY targets, shared by C1 and C2. Δ is the paired C2–C1 difference.

Harness	Model	Gold A/C	Decision Accuracy			Balanced Accuracy		
			C1	C2	Δ	C1	C2	Δ
Codex	GPT-6 Astra	312/37	0.7586	0.6418	−0.1168	0.6917	0.6129	−0.0788
Codex	GPT-5.6 Sol	312/37	0.6874	0.4712	−0.2162	0.5833	0.4167	−0.1667
Codex	GPT-5.6 Terra	312/37	0.6071	0.4417	−0.1650	0.4921	0.4557	−0.0364
Codex	GPT-5.5	312/37	0.6382	0.4527	−0.1855	0.5276	0.4378	−0.0898
Claude Code	Claude Fable 5.1	312/37	0.7837	0.6921	−0.0916	0.7184	0.6632	−0.0552
Claude Code	Claude Opus 5.5	312/37	0.7536	0.6574	−0.0962	0.6869	0.6217	−0.0652
Claude Code	Claude Sonnet 5	312/37	0.6988	0.5876	−0.1112	0.6124	0.5571	−0.0553

Table 7: RQ3 branch-specific state and clarification outcomes. Branch counts are frozen at 312 Gold-ACT and 37 Gold-CLARIFY targets in each condition; each result cell reports C1/C2.

Panel A: Gold-ACT targets

Harness	Model	Gold-ACT N C1/C2	Post-State Score C1/C2
Codex	GPT-6 Astra	312/312	0.1374/0.1038
Codex	GPT-5.6 Sol	312/312	0.0820/0.0513
Codex	GPT-5.6 Terra	312/312	0.0662/0.0219
Codex	GPT-5.5	312/312	0.0739/0.0317
Claude Code	Claude Fable 5.1	312/312	0.1578/0.1264
Claude Code	Claude Opus 5.5	312/312	0.1426/0.1097
Claude Code	Claude Sonnet 5	312/312	0.0976/0.0683

Panel B: Gold-CLARIFY targets

Harness	Model	Gold-CLARIFY N C1/C2	Blocker F1 C1/C2	Question Validity C1/C2
Codex	GPT-6 Astra	37/37	0.1127/0.0786	0.0938/0.0689
Codex	GPT-5.6 Sol	37/37	0.0574/0.0249	0.0422/0.0113
Codex	GPT-5.6 Terra	37/37	0.0401/0.0109	0.0320/0.0093
Codex	GPT-5.5	37/37	0.0478/0.0162	0.0373/0.0118
Claude Code	Claude Fable 5.1	37/37	0.1471/0.1163	0.1328/0.1076
Claude Code	Claude Opus 5.5	37/37	0.1218/0.0894	0.1067/0.0819
Claude Code	Claude Sonnet 5	37/37	0.0764/0.0491	0.0647/0.0442

At the content level, Table 7 shows that the best Post-State Score is 0.1578/0.1264, while the best Blocker F1 and Question Validity are 0.1471/0.1163 and 0.1328/0.1076. Leading decision performance therefore does not translate into complete updates or precise clarification content, and every reported branch metric declines under C2. The Phase-A bottleneck is producing the required update or identifying the precise missing information, not merely choosing whether to act.

5.3 Phase B: Repository Delivery

RQ4: Can agents translate requirements into correct code? Table 8 shows that Claude Fable 5.1 and Claude Opus 5.5 tie under C1, while Claude Fable 5.1 leads under C2. Every system passes fewer targets under C2, and none solves one quarter of the eligible targets in either condition. Executable Coverage ranges from 0.8378 to 1.0000 under C1 and from 0.8919 to 1.0000 under C2; it is unchanged or higher under C2 for six of the seven systems. The PASS Rate decline therefore reflects fewer correct deliveries rather than fewer executable evaluations.

Table 8: RQ4 repository delivery. RQ4 N is 37 in each condition; each result cell reports C1/C2. PASS Rate is computed over these RQ4 targets.

Harness	Model	RQ4 N	PASS Rate C1/C2	Exec. Coverage C1/C2
Codex	GPT-6 Astra	37	0.2162/0.1351	0.9459/1.0000
Codex	GPT-5.6 Sol	37	0.1351/0.0541	0.8649/1.0000
Codex	GPT-5.6 Terra	37	0.1081/0.0270	0.9189/0.9730
Codex	GPT-5.5	37	0.1081/0.0541	1.0000/0.9730
Claude Code	Claude Fable 5.1	37	0.2432/0.1892	1.0000/1.0000
Claude Code	Claude Opus 5.5	37	0.2432/0.1622	0.9189/1.0000
Claude Code	Claude Sonnet 5	37	0.1622/0.1081	0.8378/0.8919

6 Discussion

The results separate access to historical evidence from recovery of the state that governs the current task. Agents often retrieve supporting evidence but include unrelated or superseded material, and state reconstruction remains difficult even when relevant requirements are supplied. An agent joining a project midway therefore cannot treat the preceding conversation as a flat archive; it must determine which statements remain applicable at takeover and how subsequent events changed them.

The sharper bottleneck lies between choosing whether to act and grounding that choice in a complete requirement update. Low post-state, blocker-identification, and question-validity scores show that a correct `ACT`/`CLARIFY` decision does not by itself yield an actionable specification. Incomplete updates can carry stale constraints forward or fill unresolved fields with unsupported assumptions, whereas vague questions may fail to obtain the information needed to proceed. Reliable takeover therefore requires explicit state-delta construction, field-level ambiguity tracking, and clarification questions grounded in unresolved requirement fields.

Oracle Relevant History is not equivalent to usable context: state reconstruction, update-or-clarify reasoning, and repository delivery do not improve under C2. The condition-level comparison does not identify the cause of this decline. The audited subset may remove surrounding context needed to interpret a requirement trajectory, or the shorter input may change how the agent organizes the task even when the annotated gold evidence is preserved. History filtering should therefore preserve how requirements were introduced, revised, superseded, or resumed rather than retain only messages directly linked to the final state. Target-level paired analysis and the oracle-history sufficiency audit are needed to distinguish these mechanisms.

Repository delivery adds the final failure layer: reaching executable evaluation does not ensure behavior satisfying the frozen requirements. A useful state representation should retain the provenance of current values, represent invalidation as well as persistence, separate requirement semantics from execution evidence, and expose unresolved fields to the action policy. The frozen Phase-A response and Phase-B validator then make target-level attribution possible, distinguishing failures in history selection, state reconstruction, ambiguity handling, and implementation.

Limitations. Requirement atomicity, typed state fields, material ambiguity, and clarification targets require semantic judgment. Human review, deterministic replay, and evaluator calibration reduce this uncertainty but do not eliminate annotation or judge dependence. Multiple code changes may also satisfy the same requirement.

RQ4 contains only 37 eligible targets per condition because repository evaluation is restricted to tasks with recoverable pre-task environments and deterministically observable behavior under a frozen validator. The resulting PASS Rates and C1–C2 differences therefore have greater sampling uncertainty than the Phase-A results and should not be interpreted as broad rankings of repository-level capability. Expanding executable coverage across more projects is necessary to test whether these delivery patterns persist. In addition, multiple targets from one project are correlated, the ACT/CLARIFY classes are imbalanced, and the current aggregate results do not by themselves establish how errors propagate between stages on individual targets.

7 Conclusion

ReqMemBench evaluates coding-agent takeover as a progression from historical evidence selection to requirement-state reconstruction, update-or-clarify reasoning, and repository delivery. Across seven evaluated configurations, agents recover substantial supporting evidence but frequently over-select it, remain limited in reconstructing and updating the governing requirement state, and do not benefit from Oracle Relevant History; no system passes one quarter of the eligible RQ4 targets. These findings show that sufficient, temporally coherent context is necessary but not sufficient for reliable takeover. Before modifying a codebase, an agent must convert that context into an explicit current state and either a complete post-task specification or precise questions about unresolved requirement fields.

AI Use Disclosure

Generative AI tools assisted the privacy-preserving transformation of project histories; proposed intermediate Requirement Atoms, Events, ambiguity links, and consistency repairs during annotation; and supported code development and the model-based semantic comparisons used in evaluation. The coding agents evaluated by ReqMemBench are also generative AI systems. Generative AI tools additionally assisted manuscript drafting and readability editing. Human authors reviewed the transformed histories, corrected and approved annotations before deterministic replay and dataset freezing, inspected experimental configurations and outputs, checked the reported values against the frozen evaluation artifacts, and reviewed the final text. RQ4 outcomes were determined by frozen deterministic validators rather than generative-AI judgments. The authors take responsibility for all data transformations, annotations, experimental decisions, claims, and text in this submission.

Ethics Statement

ReqMemBench is derived from real freelance project histories and therefore carries privacy and confidentiality risks. Before benchmark construction and release, the histories pass through a multi-stage PII pipeline that detects privacy-sensitive entities, applies planned replacements or removals, verifies task-relevant semantic fidelity, and checks that identifying information is not reintroduced. The released conversations retain the message order and requirement evolution needed by the benchmark while replacing identifying surface content. Reconstructed or role-played messages and reconstructed executable environments are marked through provenance categories. Raw private conversations, credentials, direct contact details, and source artifacts not cleared for release are excluded. Human review and automated leakage checks reduce but do not eliminate re-identification risk, particularly when project details are combined with external information. The release is intended for evaluating coding-agent reasoning rather than profiling clients or workers, and users should not attempt to re-identify participants or recover private source content.

Reproducibility Statement

Sections 3–5 specify the task formulation, benchmark construction, evaluation protocol, and reported experiments, while Appendices A–C document the state schema and replay rules, human audit gates, isolation protocol, metrics, and validator calibration. The supplementary material contains the privacy-transformation and annotation code, prompts, response schemas, target-selection and instance-materialization code, frozen manifests, evaluator configurations, validators, and aggregation scripts. The released histories preserve the ordering and structural relations of the source conversations while replacing private surface content. Starting from these transformed histories, the supplied pipeline can generate fresh annotation proposals, replay Requirement State Graphs, and construct the C1/C2 RQ instances. Because model-assisted proposals and human audit are part of annotation, a fresh run is not guaranteed to reproduce the gold annotations bit for bit; the frozen annotations and manifests reproduce the benchmark used for the reported experiments.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021. URL https://arxiv.org/abs/2108.07732. MBPP.

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. LongBench: A bilingual, multitask benchmark for long context understanding. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3119–3137, 2024. doi: 10.18653/v1/2024.acl-long.172. URL https://aclanthology.org/2024.acl-long.172/.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. URL https://arxiv.org/abs/2107.03374. HumanEval.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. CrossCodeEval: A diverse and multilingual benchmark for cross-file

code completion. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, volume 36, 2023. doi: 10.52202/075280-2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/920f2dced7d32ab2ba2f1970bc306af6-Abstract-Datasets_and_Benchmarks.html.

O. C. Z. Gotel and Anthony Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of the First IEEE International Conference on Requirements Engineering*, pp. 94–101. IEEE Computer Society, 1994. doi: 10.1109/ICRE.1994.292398.

Hojae Han, Seung-won Hwang, Rajhans Samdani, and Yuxiong He. ConvCodeWorld: Benchmarking conversational code generation in reproducible feedback environments. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/6091f2bb355e960600f62566ac0e2862-Abstract-Conference.html.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. RULER: What's the real context size of your long-context language models? In *Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=kIoBbc76Sy.

Yuanzhe Hu, Yu Wang, and Julian McAuley. Evaluating memory in LLM agents via incremental multi-turn interactions. *arXiv preprint arXiv:2507.05257*, 2025. URL https://arxiv.org/abs/2507.05257. MemoryAgentBench.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/edac78c3e300629acfe6cbe9ca88fb84-Abstract-Conference.html.

Thomas Joshi, Shayan Chowdhury, and Fatih Uysal. SWE-Bench-CL: Continual learning for coding agents. *arXiv preprint arXiv:2507.00014*, 2025.

Tue Le, Minh Vu Thai Pham, Dung Nguyen Manh, Huy Nhat Phan, and Nghi D. Q. Bui. SWE-EVO: Benchmarking coding agents in long-horizon software evolution scenarios. *arXiv preprint arXiv:2512.18470*, 2025. v6 revised May 2026 – CHECK: set year to 2026 if you cite the revision or a subsequent venue version.

Tianyang Liu, Canwen Xu, and Julian McAuley. RepoBench: Benchmarking repository-level code auto-completion systems. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/d191ba4c8923ed8fd8935b7c98658b5f-Abstract-Conference.html.

Yang Liu, Li Zhang, Fang Liu, Ping Lin, and Xinyi Li. A scalable benchmark for repository-oriented long-horizon conversational context management. *arXiv preprint arXiv:2603.06358*, 2026. LoCoEval.

Pengrui Lu, Shiqi Zhang, Yunzhong Hou, Lyumanshan Ye, Chaoyi Huang, Zixi Chen, Ji Zeng, Hantao Jiang, Pengfei Liu, Yiwei Wang, and Ming-Hsuan Yang. ProjDevBench: Benchmarking AI coding agents on end-to-end project development. *arXiv preprint arXiv:2602.01655*, 2026.

Meher Bhaskar Madiraju and Meher Sai Preetam Madiraju. RigorBench: Benchmarking engineering process discipline in autonomous AI coding agents. *arXiv preprint arXiv:2606.22678*, 2026. Unrefereed preprint, 30 tasks – see notes on citation weight.

Adyasha Maharana, Dong-Ho Lee, Sergey Tulyakov, Mohit Bansal, Francesco Barbieri, and Yuwei Fang. Evaluating very long-term conversational memory of LLM agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13851–13870. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.747. URL https://aclanthology.org/2024.acl-long.747/.

Chunyu Miao, Henry Peng Zou, Yangning Li, Yankai Chen, Yibo Wang, et al. RECODE-H: A benchmark for research code development with interactive human feedback. In *International Conference on Learning Representations (ICLR)*, 2026. arXiv:2510.06186.

Samuel Miserendino, Michele Wang, Tejal Patwardhan, and Johannes Heidecke. SWE-Lancer: Can frontier LLMs earn $1 million from real-world freelance software engineering? In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 44412–44450. PMLR, 2025. URL https://proceedings.mlr.press/v267/miserendino25a.html.

Bashar Nuseibeh and Steve M. Easterbrook. Requirements engineering: A roadmap. In *Proceedings of the 22nd International Conference on Software Engineering, Future of Software Engineering Track*, pp. 35–46. ACM, 2000. doi: 10.1145/336512.336523.

Gabriel Orlanski, Devjeet Roy, Alexander Yun, Changho Shin, Alex Gu, Albert Ge, Dyah Adila, Nicholas Roberts, Frederic Sala, and Aws Albarghouthi. SlopCodeBench: Benchmarking how coding agents degrade over long-horizon iterative tasks. *arXiv preprint arXiv:2603.24755*, 2026.

Di Wu, Hongwei Wang, Wenhao Yu, Yuwei Zhang, Kai-Wei Chang, and Dong Yu. LongMemEval: Benchmarking chat assistants on long-term interactive memory. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/d813d324dbf0598bbdc9c8e79740ed01-Abstract-Conference.html.

Zexun Zhan, Shuzheng Gao, Ruida Hu, and Cuiyun Gao. SR-Eval: Evaluating LLMs on code generation under stepwise requirement refinement. *arXiv preprint arXiv:2509.18808*, 2025.

Jiayuan Zhu, Junde Wu, Minhao Hu, Shengda Zhu, Jiazhen Pan, Weixiang Shen, Yijun Yang, Fenglin Liu, Jianye Hao, Yueming Jin, Qirong Ho, and Min Xu. SWE Context Bench: A benchmark for context learning in coding. *arXiv preprint arXiv:2602.08316*, 2026. Official name is "SWE Context Bench"; not the same paper as ContextBench (arXiv:2602.05892).

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, et al. BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. In *International Conference on Learning Representations (ICLR)*, 2025. arXiv:2406.15877.

A FORMAL STATE SPECIFICATION

This appendix specifies the state representation summarized in Section 3. A Requirement Atom is the smallest independently evolving requirement. Table 9 summarizes its five dimensions, whose values are maintained separately rather than collapsed into a single status.

A.1 STATE DIMENSIONS

Table 9: Canonical dimensions of a Requirement State. Unknown values remain explicit and do not receive inferred defaults.

Dimension	Role	Representation
A_r (attributes)	Requirement-specific values	Typed key–value leaves such as thresholds, amounts, policies, and expected behavior. Unobserved values remain absent or unknown.
S_r (scope)	Where and when the requirement applies	Persistence, affected components, and contextual scopes.
L_r (lifecycle)	Whether the requirement currently governs the project	`ACTIVE`, `DEFERRED`, `REMOVED`, or unknown when no complete baseline is observable.
U_r (ambiguity)	Unresolved uncertainty	Zero or more open records identifying the affected dimension, field when available, missing information, and source Event.
X_r (execution)	Latest implementation evidence	`UNKNOWN`, `CLAIMED_WORKING`, `FAILED`, or `VERIFIED_WORKING`, with the observation and source Event when available.

Lifecycle, ambiguity, and execution evidence are independent. An active requirement may remain ambiguous or have failed execution evidence. Deferral and removal preserve the last known attributes and scope, while lifecycle determines whether those values currently apply. Removed requirements remain in the graph so that later agents can distinguish invalidated requirements from information that was never observed.

If the observable history begins after a requirement was introduced, replay starts from an incomplete observed baseline. Unknown fields remain unknown, and the earlier Events are retained. A later explicit `INTRODUCE` establishes a complete baseline without deleting the preceding observations.

A.2 EVENTS AND REPLAY

Each reviewed Event changes one part of the preceding state. Table 10 gives the canonical effects. An Event closes an ambiguity only when its `resolves_ambiguity_event_ids` field explicitly names that ambiguity; temporal proximity or Event type alone never implies resolution.

Replay follows normalized conversation order, not the numeric order of message identifiers. Before replay, validation checks identifier uniqueness, Event payloads, source-message references, ambiguity-resolution links, and ordering constraints. A Requirement may contain at most one `INTRODUCE`, and no later Event may follow `REMOVE`. Any violated invariant stops construction instead of triggering model-based repair. Every replayed State retains the supporting Event identifiers needed to trace its fields back to pre-target evidence.

A.3 STATE GRAPHS AND TARGET BOUNDARIES

Each Event creates a new node in the Requirement-level State Graph, and the project graph collects these trajectories without merging independently evolving atoms. A selected client message defines one target even when it produces several Events. All Events supported by that message are applied in their reviewed order.

Table 10: Requirement Event transitions. "Preserve" means that the previous value remains in the next State.

Event	State update	Preserved information
INTRODUCE	Initializes attributes and scope and sets lifecycle to ACTIVE; clears execution evidence from an incomplete observed baseline.	Open ambiguities except those closed by explicit links.
MODIFY	Updates or removes named attributes or scope fields and clears execution evidence for the superseded semantic version.	Lifecycle and unmodified fields.
DEFER	Sets lifecycle to DEFERRED.	Attributes, scope, ambiguity, and execution evidence.
RESUME	Sets lifecycle to ACTIVE.	Attributes, scope, ambiguity, and execution evidence.
REMOVE	Sets lifecycle to REMOVED.	Prior attributes, scope, and execution evidence.
AMBIGUOUS	Adds an open ambiguity without guessing a value or changing lifecycle.	Attributes, scope, lifecycle, and execution evidence.
IMPLEMENTATION_CLAIM	Sets execution to CLAIMED_WORKING.	Requirement semantics and lifecycle.
RUNTIME_FAILURE	Sets execution to FAILED.	Requirement semantics and lifecycle.
RUNTIME_VERIFICATION	Sets execution to VERIFIED_WORKING.	Requirement semantics and lifecycle.

The pre-task snapshot contains the last State strictly before the target message. The post-task snapshot applies every Event supported by the target and stops before later implementation or runtime evidence. A requirement introduced by the target is absent from the pre-task snapshot, while a removed requirement remains in the post-task snapshot with lifecycle REMOVED. The stored snapshots cover the project; evaluation uses the task-local projection defined in Equation 3.

B Benchmark Construction and Curation Audits

B.1 Annotation and Target Selection

Construction begins from privacy-preserving project timelines. Annotators identify Requirement Atoms and record only evidence-supported Events, including explicit links between an ambiguity and the later Event that resolves it. Model-generated proposals accelerate this process, while schema checks reject malformed objects and human review freezes the semantic annotation. Each instance carries a coarse provenance category for transformed records, reconstructed messages, recovered environments, or reconstructed environments. These categories support aggregate reporting without exposing raw participant records or identifiable project artifacts.

A target candidate is one client message together with all Requirement Events supported by that message. Its history contains only earlier messages. The selector retains candidates that describe a valid task, depend on prior project context, and exercise requirement evolution. Human review accepts, rejects, or restores candidates after checking the historical-dependence judgment and coverage of Event and task types. Context-free introductions and messages that only report implementation evidence are excluded.

For an accepted target, deterministic replay produces complete pre- and post-task project snapshots. The instance builder expands the affected State references, separates historical requirements from requirements introduced by the target, and derives the applicable evaluation stages. RQ1 and RQ2 require at least one affected historical requirement. RQ3 requires at least one affected transition supported by the target. An RQ4 candidate additionally requires a matching pre-task code environment; formal RQ4 eligibility is determined only after the RQ3 gold decision and execution checks are frozen.

B.2 History Conditions and Leakage Controls

C1 contains every admissible pre-target message. C2 contains the ordered union of the affected requirements' pre-target trajectories and the contextual messages needed to interpret them. No-History is not a benchmark condition. RQ1 is evaluated only under C1, whereas RQ2–RQ4 use both conditions when applicable.

Phase-A inputs contain only the current task, the condition-specific history, common instructions, and the public response schema. They exclude internal Requirement, Event, and State identifiers; target position and condition labels; gold annotations; future messages; repository paths and files; Acceptance Criteria; judge configurations; and evaluator metadata. The response is stored immutably with its hash before Phase B. For eligible execution runs, a fresh copy of the same pre-task repository is then opened in an isolated workspace. C1 and C2 never share sessions, scratch state, or modified repositories, and the Phase-B repository cannot be used to revise the frozen Phase-A response.

Before release, each public package is checked against its declared message set. C2 must be an ordered subsequence of C1; every history message must precede the target; and the target itself may appear only in the task file. RQ4 repositories undergo separate audits for future-state, Acceptance-Criteria, judge-asset, and evaluator-metadata leakage.

B.3 Human Audit Procedure

Human review is applied at three construction gates. During State-Graph construction, reviewers compare model-proposed Requirement boundaries, Event types and payloads, source-message links, lifecycle changes, and ambiguity-resolution links against the transformed project timeline. Unsupported annotations are corrected before deterministic replay, and an instance cannot advance while schema, source-link, ordering, or replay invariants fail.

During target construction, reviewers verify that the selected client message defines a valid task, depends on earlier project context, and has a complete affected-Requirement and Event set. They then inspect the replayed pre- and post-task snapshots, including whether the target introduces a requirement, changes an existing state, or leaves a material ambiguity that requires clarification. Borderline candidates are accepted, rejected, or restored only after this review.

During condition and execution review, reviewers verify that C2 is an order-preserving subsequence of C1 and retains the requirement trajectories and contextual messages needed to support the same gold pre-task State and RQ3 branch. They also inspect the packaged inputs for future-message, gold-annotation, repository, and evaluator leakage. For RQ4 candidates, review checks the Gold-ACT decision, the temporal match of the pre-task repository, the observability of the requested change, and the validator package before the target is admitted to executable evaluation. Human-reviewed artifacts are frozen only after the corresponding deterministic checks pass.

C Evaluation Protocol and Metrics

C.1 Run Unit and Isolation

The logical run unit is a target–condition pair. Phase A produces one structured response containing Requirement selection, pre-task States, and the update-or-clarify decision. The runner validates and freezes that response before determining whether Phase B may start. Every condition and replicate receives a new process and workspace; model configuration, prompt, non-repository tools, and resource limits are identical across the paired C1/C2 runs.

RQ1–RQ3 runs never receive repository access. A Phase-A response must pass parsing and schema validation before it is frozen. Phase B starts only when the condition is RQ4-eligible and the valid frozen agent decision is `ACT`. On a gold-eligible target, a valid non-`ACT` decision produces `NO_CODE_SUBMISSION` and an RQ4 failure. A Phase-A process, parsing, schema, or harness failure leaves the attempt incomplete and triggers a clean rerun. After Phase B starts, an agent run that does not leave a testable frozen repository is an RQ4 failure, whereas an environment, validator, or evaluator failure marks the attempt as `INVALID_ATTEMPT` and also triggers a clean rerun.

C.2 RQ1 and RQ2

For RQ1, the gold set consists of affected requirements that existed before the target. A frozen semantic classifier labels every predicted–gold pair using a discrete relation vocabulary. Only `SAME_ATOM` creates an alignment edge; merged, partial, merely related, unrelated, and uncertain pairs do not match. Deterministic maximum-cardinality, maximum-weight one-to-one matching

then produces Requirement true positives, false positives, and false negatives. We report Requirement Precision, Recall, and F1.

Gold evidence is organized into groups, one for each current-support Event, with alternative source messages allowed within a group. A deterministic one-to-one match between predicted message claims and gold groups produces Evidence Precision, Recall, and F1. Repeated alternatives for the same group do not create additional true positives. Messages retained as neutral trajectory context are recorded diagnostically but do not count as evidence false positives. Requirement and Evidence metrics remain separate and are macro-averaged over targets.

RQ2 scores only successfully aligned historical requirements. For target t, let M_t be the one-to-one matched requirements and R_t^{gold} the historical gold set. Coverage is

$$\text{ReconstructionCoverage}_t = \frac{|M_t|}{|R_t^{\text{gold}}|}. \tag{5}$$

A target with no match is `N/A` for matched-state quality, not a perfect reconstruction.

Each scorable State leaf has a frozen comparator. Booleans, enums, identifiers, and exact numbers use normalized deterministic comparison; unordered collections use set F1; ordered sequences use order-sensitive comparison; and nested objects recurse to scorable leaves. Free-text facts use a frozen semantic-equivalence classifier, while unknown or unreliable gold leaves are skipped. Attribute keys are aligned before values are scored so that semantically equivalent field names may match without relaxing numeric or categorical correctness. If P_r and G_r are the predicted and gold attribute leaves, M_r their one-to-one alignment, and $v(p, g) \in [0, 1]$ the typed value score, then

$$\text{AttributeScore}(r) = \frac{2 \sum_{(p,g)\in M_r} v(p, g)}{|P_r| + |G_r|}. \tag{6}$$

Extra stale fields and missing fields therefore reduce the closed-world score. We report Attribute Reconstruction and Reconstruction Coverage.

C.3 RQ3 AND RQ4

RQ3 first evaluates the `ACT`/`CLARIFY` decision. Decision Accuracy and Balanced Accuracy are reported with the Gold-ACT and Gold-CLARIFY counts. For Gold-ACT targets, every affected requirement, including those introduced by the target, must be aligned and assigned a complete post-task State. Missing requirements score zero, and extra requirements are closed-world false positives.

For Gold-CLARIFY targets, each blocking issue is represented by its requirement, uncertainty dimension, optional field, and missing information. One-to-one issue matching yields the reported Blocking-Issue F1. Question Validity measures whether a question asks for the missing information without assuming an answer.

RQ4 evaluates only the final repository. A target–condition pair is eligible only when the gold RQ3 decision is `ACT`, the pre-task environment is reproducibly runnable, the requested change is material and deterministically observable, the Acceptance Criteria and target-specific validator package are frozen, the pre-task repository passes the calibration in Section C.4, and the repository has passed leakage review. For a scored attempt, the binary outcome is

$$\text{RQ4Pass} = \text{BuildPass} \wedge \text{TargetTestPass} \wedge \text{RegressionPass}. \tag{7}$$

The three components respectively test deliverable validity, the requested behavioral change, and preservation of behavior that should remain unchanged. A local finalizer recomputes the conjunction from their machine-readable records rather than accepting a validator-reported overall verdict. An explicit failure in any component produces `FAIL`; all three components must pass for `PASS`. Validator timeouts, malformed result records, missing component results, and environment or tool failures produce `INVALID_ATTEMPT` with no binary result and are rerun before formal inclusion.

PASS Rate is the number of passing attempts divided by the eligible attempts with a valid `PASS` or `FAIL` result. We separately report

$$\text{ExecutableCoverage} = \frac{\#\text{ target–condition pairs receiving PASS/FAIL}}{\#\text{ RQ4-eligible target–condition pairs}}. \tag{8}$$

RQ4 reports PASS Rate and Executable Coverage separately for C1 and C2.

C.4 Aggregation and Validator Calibration

Metrics are computed at their declared unit, macro-averaged over targets, and then macro-averaged over projects for benchmark-level claims. Paired C1/C2 comparisons include only targets with identical gold and valid results in both conditions. Semantic-judge, response-schema, validator, and harness infrastructure failures are recorded separately and rerun.

RQ4 uses deterministic validators rather than a model-based judge. Each target-specific validator is run on the frozen pre-task repository and must produce `Build=PASS`, `TargetTest=FAIL`, and `Regression=PASS`. This check establishes that the starting repository is runnable, the target test detects the requested change before implementation, and the preservation checks hold before the agent acts. Of 40 validator proposals, 37 satisfy this contract; three are excluded because their target tests already pass on the pre-task repository. The formal registry freezes the included target identities, validator packages, commands, timeouts, and aggregation code. Reference implementations and partial or mutant deliveries are created, and additional independent review is conducted, only when validator feasibility or coverage is uncertain; none is a per-target prerequisite. A later change to a frozen validator or aggregation rule creates a new evaluator version and requires the affected runs to be repeated.