REPOTRACE: IMPROVING REQUIREMENT TRACING IN MULTI-ROUND AGENT INTERACTIONS

Anonymous authors
Paper under double-blind review

ABSTRACT

Code agents have become increasingly capable of performing software engineering tasks through interaction with software environments. However, agents tend to degrade as the number of interaction rounds increases, making it difficult to maintain existing software during iterative modification. We analyze 2,656 agent trajectories in EvoCode-Bench and identify **requirement focus misalignment** as a recurring failure pattern, where agents struggle to distinguish the primary modification targets of a requirement from unrelated code.

We propose **RepoTrace**, a requirement-driven repository understanding harness that combines repository structure with requirement semantics to construct a requirement-code correspondence and injects it into the agent context. This correspondence enables code agents to follow the repository architecture, trace requirements across multi-round interactions, and identify the core code elements corresponding to the current requirement. Experiments show that **RepoTrace** improves the mean per-round reward (MPR) across different models (*e.g.,* +19.99 on GLM-5.2) and agent harnesses (*e.g.,* +10.22 on Claude Code). Across rounds, **RepoTrace** increases the average per-round reward from 0.163 to 0.371 on Terminus-2 with DeepSeek-V4-Pro, while reducing average regression errors by 56.78%. We release the source code and agents' trajectories to support future research[1].

1 INTRODUCTION

Recently, code agents have become increasingly capable of performing software engineering tasks through interaction with software environments (Anthropic, 2026; OpenAI, 2026). A harness enables code agents to observe the environment, use tools, execute actions, receive execution feedback, and update memory (Sumers et al., 2024). Through multi-round interaction with the environment, code agents have demonstrated strong capabilities in repository-level code generation, vulnerability discovery, and software testing (Anthropic, 2026; Carlini et al., 2026; Maaz et al., 2026). Vibe coding shifts software development from one-shot code generation to iterative modification of existing software systems (Tang et al., 2026), where software maintenance and evolution are also important parts of the software life cycle (Lehman, 1980; Lientz et al., 1978). Future software development requires agents not only to satisfy current user requirements, but also to understand the architecture of existing repositories and the potential impact of their modifications.

However, recent studies show that code agents tend to degrade as the number of interaction rounds increases in complex multi-round software engineering tasks (Shen et al., 2026; Huang et al., 2026; Orlanski et al., 2026; Laban et al., 2025). For example, EvoCode-Bench (Shen et al., 2026) contains 26 stateful tasks and 227 interaction rounds from 12 mainstream code agents. The analysis of failure cases shows that the test pass rate decreases from 46.7% in early rounds to 7.7% in later rounds. Regression Accumulation (Huang et al., 2026) evaluates 542 tasks over 8 rounds, covering 26,016 interaction instances, and reports that 40%–73% of tasks lose previously satisfied behaviors during multi-round interactions.

Software has two fundamental properties: constructivity and evolvability (Yang, 2005; Breivold et al., 2012). Constructivity refers to building software systems that satisfy functional goals based on requirements. Evolvability refers to the ability of software systems to iteratively adapt to changes in

[1] https://anonymous.4open.science/r/RepoTrace-anonymous-1112

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.

requirements, environments, and technologies while maintaining system integrity and consistency. The challenges introduced by software evolvability receive more and more attention, including architecture maintenance and dependency management (Gu et al., 2026). Recent studies have explored self-evolving software and recursive self-improvement (RSI), further demonstrating the importance of evolvability (Zhang et al., 2026; Liu et al., 2026a; Zhou, 2026).

To further understand the challenges faced by code agents during multi-round interactions, we systematically analyze 2,656 agent trajectories from EvoCode-Bench. We identify **requirement focus misalignment** as a recurring failure pattern. Specifically, code agents struggle to distinguish the primary modification targets of a requirement from related code elements (*e.g.,* classes, functions, and files) that do not require direct modification. As a result, code agents may modify unrelated code or miss necessary synchronized updates. Across failure rounds, 55.16% of code modifications occur outside the modification target files specified by the human oracle (see Section 2).

To this end, we propose **RepoTrace**, a requirement-driven repository understanding harness that improves requirement tracing for code agents in multi-round software tasks. The core idea of **RepoTrace** is to explicitly guide code agents to establish the correspondence between requirements and the code repository before code modification. Specifically, **RepoTrace** first builds a structural representation of the code repository from code relations and computes a structural weight for each code element. The weights help identify core elements, extension points, and change propagation across the repository. Next, **RepoTrace** extracts the potential modification scope from the semantics of the natural-language requirement. Finally, **RepoTrace** combines requirement semantics with repository structure to construct a requirement-code correspondence and injects it into the agent context. **RepoTrace** allows code agents to trace the repository architecture across rounds and identify the core code elements corresponding to the current requirements.

We apply **RepoTrace** to multiple mainstream code agents and conduct experiments on multi-round software tasks. Experimental results show that: ❶ Across models, **RepoTrace** improves the mean per-round reward (MPR) by +19.99 on GLM-5.2 and +4.91 on MiniMax-M3. ❷ Across harnesses, **RepoTrace** improves MPR by +22.25 on Terminus-2 and +10.22 on Claude Code. ❸ Across rounds, **RepoTrace** increases the average per-round reward from 0.163 to 0.371 on Terminus-2 with DeepSeek-V4-Pro, while reducing average regression errors by 56.78%. The main contributions are summarized as follows:

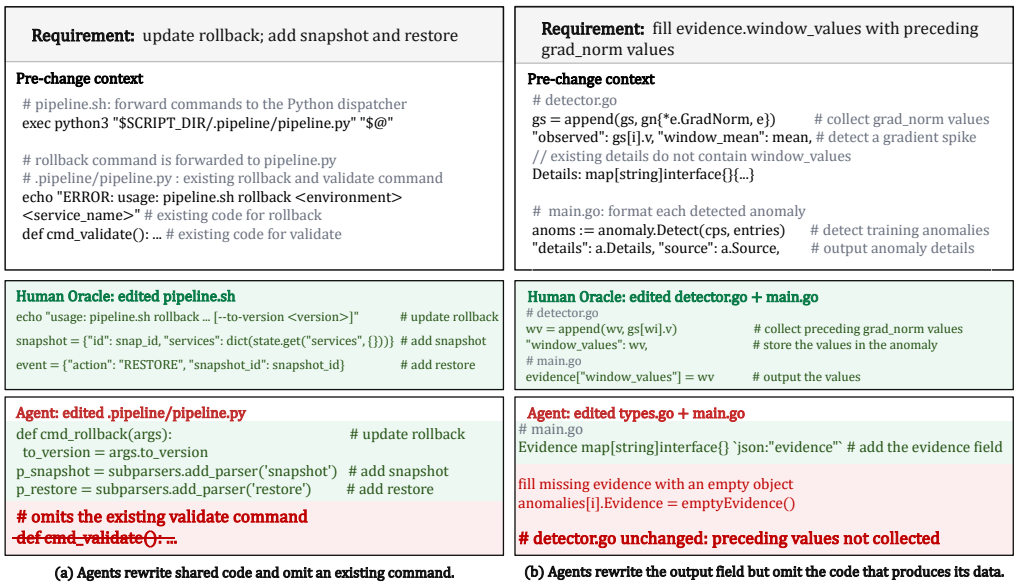
1) We analyze agent trajectories and identify **requirement focus misalignment** as a recurring failure pattern during multi-round interactions.
2) We propose **RepoTrace**, a requirement-driven repository understanding harness that combines repository structure and requirement semantics to improve requirement tracing for agents.
3) Experimental results show that **RepoTrace** improves performance across models and harnesses in multi-round interactions, and reduces regression errors.

2 MOTIVATING EXAMPLES

In this section, we use two real-world examples of multi-round repository modification to illustrate **requirement focus misalignment**: code agents modify unrelated code or omit necessary updates. The examples are from real-world software development tasks in EvoCode-Bench (Shen et al., 2026) and include human-written oracle solutions.

❶ **Unrelated code modification.** In Figure 1(a), the requirement asks the agent to update the `rollback` behavior and add the `snapshot()` and `restore()` functions. The human-written oracle modifies the corresponding logic in `pipeline.sh`. The agent instead rewrites `.pipeline/pipeline.py`, a shared command dispatcher that contains commands unrelated to the requirement. During the rewrite, the agent omits the existing `validate()` command and breaks the validation functionality.

❷ **Missing update.** In Figure 1(b), the requirement asks each detected anomaly to include an `evidence` object. For a gradient spike, `evidence.window_values` records up to five preceding gradient-norm values. The values are collected in `detector.go` and formatted for output in `main.go`. The human-written oracle updates both components. However, the agent modifies



(a) Agents rewrite shared code and omit an existing command.

(b) Agents rewrite the output field but omit the code that produces its data.

Figure 1: Two examples of **requirement focus misalignment**. (a) Unrelated code modification: the agent rewrites a shared dispatcher and omits the existing `validate()` command. (b) Missing update: the agent adds the output field but omits the code that collects its values.

only the output-related code in `types.go` and `main.go` and omits the update to `detector.go`. Consequently, the output contains the `evidence` field, but `window_values` remains empty.

The two examples show two forms of **requirement focus misalignment**: unrelated code modifications and missing updates. Across failed interaction rounds after Round 1, 55.16% of source-file modifications occur outside the modification target files specified by the human-written oracle (see Appendix B for the experimental details). The finding motivates **RepoTrace**, which helps code agents identify the code elements corresponding to the current requirement.

3 Related Work

3.1 Code Agents in Multi-Round Interactions

Code agents have demonstrated strong capabilities in software engineering tasks. Commercial products such as Claude Code, Codex and Kimi Code (Anthropic, 2026; OpenAI, 2026; Moonshot AI, 2026) bring agentic coding into daily development practice. From function-level HumanEval (Chen et al., 2021), to repository-level SWE-bench (OpenAI, 2024; Deng et al., 2025), to long-horizon tasks like Terminal-Bench (Merrill et al., 2026), code agents continuously achieve high scores. Many existing benchmarks focus on single-round, one-shot code generation, where software maintenance and evolution remain important throughout the software life cycle (Lehman, 1980; Lientz et al., 1978). However, recent studies show that code agents tend to degrade as the number of interaction rounds increases. Laban et al. (2025) finds that top-tier models lose roughly 39% of their average success rate compared to single-round settings, and Rawal et al. (2025) reports a 20%-27% average drop in correct-and-secure outputs across 32 models. Similarly, SlopCodeBench reports that no agent solves all tasks, where the best agent passes only 14.8% of checkpoints, and 77% of trajectories exhibit structural erosion, while 75.5% suffer from verbosity (Orlanski et al., 2026). Complementary studies report consistent findings, including declining accuracy under iterative refinement, low completion rates under stepwise requirement refinement, instruction forgetting, and progressive faithfulness loss to emergent specifications (Wang et al., 2026; Zhan et al., 2025; Wang et al., 2025; Yan et al., 2026).

3.2 Requirements Tracing and Repository-Level Code Understanding

Requirements traceability refers to the ability to describe and follow the life of a requirement (Gotel & Finkelstein, 1994), which is an active research topic in software engineering. Recent studies

explore the combination of LLMs with structured methods for requirements tracing. TraceDev proposes a multi-agent framework that links requirements, design models, and code in a heterogeneous traceability graph to explicitly maintain consistency among software artifacts (Chen et al., 2026). RAG-based methods further recover requirement trace links in a largely domain-agnostic manner (Hey et al., 2025; Fuchß et al., 2025). Existing studies demonstrate the potential of language models in requirements tracing and the impact analysis of change (Etezadi et al., 2025). However, they don't consider how requirement-artifact consistency should be maintained as requirements evolve over multiple rounds of development.

Meanwhile, repository-level code understanding is another key research direction. GraphCoder proposes the code context graph (CCG), which structures information such as control flow and data dependencies into a graph to improve context retrieval for repository-level code completion (Liu et al., 2024). CodexGraph and RANGER organize repositories as knowledge graphs, supporting graph queries and natural-language retrieval (Liu et al., 2025; Shah et al., 2025). Most recently, LocAgent and DyRetriever allow agents to navigate or perform multi-hop reasoning along dependency graphs, achieving low-cost code localization and context retrieval (Chen et al., 2025; Liu et al., 2026b). Existing methods enhance the understanding of code structure from different angles (Luo et al., 2025). However, they primarily target static repositories or one-shot generation, without considering how repository structure should be maintained and updated during multi-round interactions.

4 REPOTRACE

We propose **RepoTrace**, a requirement-driven repository understanding harness for code agents in multi-round interactions. To address requirement focus misalignment, **RepoTrace** combines repository structure with requirement semantics before code modification. The agent receives the current requirement together with structural dependencies and information about code elements associated with prior requirements.

Figure 2 illustrates the framework. At round t, **RepoTrace** takes the current code repository R_t and the natural-language requirement r_t. The following subsections introduce three steps: ❶ how **RepoTrace** constructs a structural representation G_t of the repository; ❷ how **RepoTrace** extracts requirement entities Q_t using the current requirement, the structural representation G_t, and prior requirement records; and ❸ how **RepoTrace** combines G_t and Q_t to construct a requirement-code correspondence message M_t, maintain requirement tracing across rounds.

4.1 REPOSITORY STRUCTURE ANALYSIS

Structural representation. Let V_t denote the set of source files collected from R_t, and let E_t denote the collected directed file relations. A relation $(u, v) \in E_t$ indicates that file u refers to code in file v. The representation G_t contains these file and relation sets, an index that associates symbols with their defining files, and file-level structural weights. To obtain G_t, we first extract the relations and symbol index, then compute the structural weights. We also compare G_t with the previous representation to identify structural changes across rounds.

Code relations. To obtain the file relations and symbol index, **RepoTrace** collects source-file paths, import declarations, symbol definitions, and candidate call sites from R_t. Specifically, we record (u, v) when file u imports code from file v or contains a candidate call site whose name matches a symbol defined in v. We resolve repository-local imports and aggregate matched calls into file relations. These relations provide an approximate structural view, and the symbol index connects names in a requirement to repository files.

Structural weights. To identify core files that may affect dependent files, we compute two structural scores from code relations. We apply PageRank (Page et al., 1999) to the file graph formed by nodes V_t and relations from E_t whose endpoints are both in V_t. We also compute the fan-in of each file, defined as the number of distinct files that refer to it. PageRank captures the structural importance of files in the dependency graph, while fan-in measures how many files directly depend on each file. We use PageRank to prioritize files included in the LLM prompt and to order examples of dependent files. We use fan-in to select files referenced by multiple files and to rank the existing-file

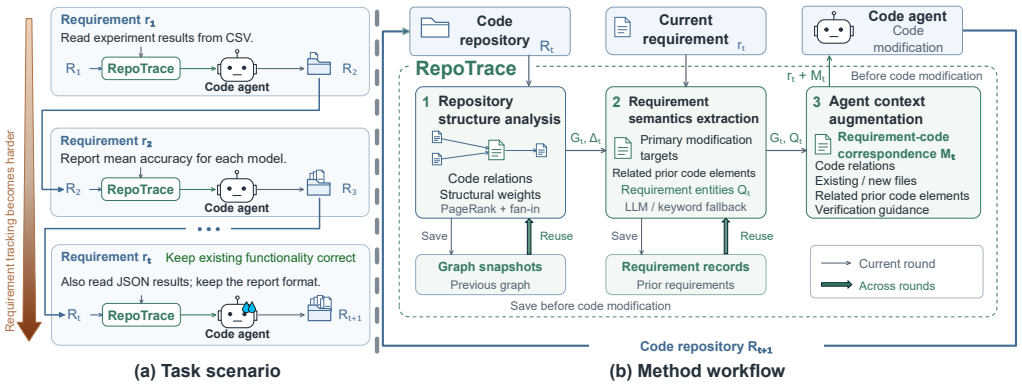


Figure 2: Overview of **RepoTrace** in multi-round interactions. Left: an illustrative sequence of requirements for the same code repository. Right: before each code modification, **RepoTrace** analyzes repository structure, extracts requirement semantics, and constructs a requirement-code correspondence message M_t for the code agent.

overview in the agent context. Two scores expose structural dependencies **without** assuming that structurally important files require modification.

Structural changes. To reflect code modifications across rounds, **RepoTrace** reconstructs G_t from the current repository before processing each requirement. To identify what has changed, we compare G_t with the most recent retained representation G_{prev}, whose file and relation sets are V_{prev} and E_{prev}. We summarize structural changes as

$$\Delta_t = \big(V_t \setminus V_{\text{prev}},\ V_{\text{prev}} \setminus V_t,\ E_t \setminus E_{\text{prev}},\ E_{\text{prev}} \setminus E_t\big). \tag{1}$$

The four components record added files, removed files, added relations, and removed relations, respectively. For the first round, the previous file and relation sets are empty. This summary provides structural changes observed before the current modification, including changes made during preceding rounds. The analysis produces the current structural representation G_t and the change summary Δ_t for requirement semantics extraction. For specific examples, see Appendix A.1.

4.2 Requirement Semantics Extraction

Building on G_t and Δ_t, **RepoTrace** analyzes the current requirement r_t and prior requirements $r_{<t}$, in order to identify candidate primary modification targets and potentially affected code elements. To connect r_t with $r_{<t}$, **RepoTrace** retains a history $\mathcal{H}_{<t}$ of earlier requirement records, including their extracted code elements and summaries. **RepoTrace** combines r_t, G_t, Δ_t, and $\mathcal{H}_{<t}$ to produce a structured record of requirement entities:

$$Q_t = \text{Extract}\big(r_t, G_t, \Delta_t, \mathcal{H}_{<t}\big). \tag{2}$$

The record Q_t contains candidate target files and symbols, behavior constraints, potentially affected prior code elements, and a requirement summary. These fields represent two roles: candidate primary modification targets and potentially affected prior code elements.

❶ For LLM-based extraction, **RepoTrace** calls an LLM to analyze the current requirement, a list of files ranked by PageRank, symbol-to-file associations, and a bounded summary of recent requirement records. When prior records exist, **RepoTrace** also includes a compact summary of Δ_t in the prompt. The LLM extracts the requirement entities and returns the structured record Q_t. ❷ If no LLM is configured or extraction fails, a keyword fallback matches file paths and identifiers in r_t against the repository index. The fallback compares these matches with historical targets to distinguish newly identified targets from potentially affected prior code elements without using Δ_t. In both paths, the records retain associations between requirements and code elements across rounds. For specific examples, see Appendix A.2.

4.3 Agent Context Augmentation across Rounds

The requirement semantics extraction produces Q_t, which identifies candidate primary modification targets and potentially affected prior code elements. **RepoTrace** combines this requirement-specific

information with the structural representation G_t to construct a requirement-code correspondence message for the code agent. To maintain requirement tracing across rounds, **RepoTrace** also retains the current structural representation and requirement record for use in subsequent rounds.

Requirement-code correspondence. The structural representation G_t describes code relations, while Q_t identifies candidate code elements associated with the current and prior requirements. To provide the code agent with context for the current requirement, we select information from both to construct the requirement-code correspondence message:

$$M_t = \mathrm{Compose}(G_t, Q_t), \qquad \widetilde{r}_t = r_t \circ M_t, \tag{3}$$

where $\circ$ denotes text concatenation. The message combines four types of information: code relations for files with high fan-in and selected direct dependents; an overview of existing files and candidate new files; potentially affected prior code elements; and verification guidance. We identify candidate new files by comparing the target paths in Q_t with V_t. The verification guidance asks the agent to build and test the modified repository and suggests a command based on the programming languages in the repository. The message uses selected fields of Q_t, rather than reproducing the entire extraction record. The code agent receives the original requirement with this supplemental context and retains responsibility for code modification and verification.

Maintaining records across rounds. To reuse the current analysis in later rounds, **RepoTrace** stores the current graph representation and appends the current requirement record to its history after constructing M_t and before the code agent executes. The agent then modifies the repository, whose state is observed as R_{t+1} at the next round. At that round, **RepoTrace** reconstructs the repository structure, compares it with the retained graph, and reuses prior requirement records during extraction, repeating the three steps for the next requirement. Graph snapshots and requirement records therefore describe prior analyses, rather than verified completion states. For specific examples, see Appendix A.3.

5 Experiments

To evaluate the effectiveness of **RepoTrace**, we conduct experiments with multiple code agents on multi-round software tasks. In this section, we introduce the benchmarks, baselines, models, and evaluation metrics. We then present the experimental results and analysis.

5.1 Experimental Setup

Benchmarks. We evaluate code agents on EvoCode-Bench (Shen et al., 2026), which contains 26 stateful software engineering tasks and 227 evaluated rounds across six technical domains, including ML/MLOps, data engineering, systems and code, scientific computing, testing and automation, and infrastructure and security. Each task contains 5-15 rounds, with the same workspace and agent session preserved across rounds. Each round introduces a new requirement and uses cumulative test cases to verify the current and previous active requirements. A round receives a reward only when all test cases pass.

Baselines and Models. We integrate **RepoTrace** into two agent harnesses, Terminus-2 (Harbor Framework Team, 2026b) and Claude Code (Anthropic, 2026), within the Harbor agent environment (Harbor Framework Team, 2026a). We evaluate four models: GLM-5.2 (GLM, 2026), DeepSeek-V4-Pro (DeepSeek-AI, 2026), MiniMax-M3 (Lai et al., 2026), and Qwen3.7-Max (Qwen Team, 2026). We access all models via API. All API configurations (*e.g.,* temperature, top-k, and thinking effort) use the default settings.

Evaluation Metrics. We follow the evaluation setting of EvoCode-Bench (Shen et al., 2026).

1) **Mean per-round reward (MPR)** $\uparrow$ measures the fraction of passed rounds, where a round is counted as passed only when all test cases pass. The reward is averaged across all tasks.
2) **Final-round completion rate (FCR)** $\uparrow$ measures the average test case completion rate of the final round across all tasks. The test cases are cumulative, so completing the final-round test cases means that all requirements in the requirement chain are satisfied.

Table 1: Main results and token usage on EvoCode-Bench.

	MPR ↑	FCR ↑	PTR ↑	HTR ↓	TCS ↑	IT(M) ↓	OT(M) ↓
DeepSeek-V4-Pro	14.14	11.54	3.85	92.31	61.74	441.33	**12.51**
w/ RepoTrace	**36.39**	**34.62**	**15.38**	**61.54**	**73.63**	**319.48**	16.92
GLM-5.2	16.19	3.85	3.85	88.46	47.52	**98.26**	**7.62**
w/ RepoTrace	**36.18**	**30.77**	**7.69**	**61.54**	**77.33**	175.62	12.30
MiniMax-M3	23.42	**23.08**	**7.69**	**73.08**	61.46	**726.66**	**5.89**
w/ RepoTrace	**28.33**	15.38	**7.69**	**73.08**	**76.76**	1445.57	7.99
Qwen3.7-Max	**11.87**	**7.69**	0.00	**92.31**	**67.37**	**227.49**	**7.89**
w/ RepoTrace	11.36	3.85	0.00	**92.31**	66.06	275.64	9.07

3) **Perfect-task rate (PTR)** ↑ measures the fraction of tasks in which a single attempt passes every round.
4) **Hard-task rate (HTR)** ↓ measures the fraction of tasks whose per-task score is at most 0.5.
5) **Test Case Score (TCS)** ↑ measures the average test case completion rate of the entire dataset.
6) **Input / Output Tokens (IT / OT)** ↓ measure the total number of input / output tokens used across all rounds of all tasks in the dataset, expressed in millions of tokens (M).

5.2 RQ1: To what extent does **RepoTrace** improve performance across different models?

To answer RQ1, we evaluate four representative models with and without **RepoTrace** on EvoCode-Bench, in order to verify whether **RepoTrace** improves model performance on multi-round software engineering tasks. Table 1 reports the results.

RepoTrace brings improvement on stronger models that can effectively exploit repository-level guidance, while the benefit is limited for models that struggle to utilize it. On DeepSeek-V4-Pro, **RepoTrace** raises MPR from 14.14 to 36.39 (+22.25, about $2.57\times$), with FCR and PTR improving by about $3\times$ and $4\times$ respectively. This shows that the traced repository-level guidance translates into both higher per-round success and more fully completed tasks. The benefit is limited for Qwen3.7-Max, suggesting that the advantage scales with the model's ability to exploit the injected guidance. The extra token cost is acceptable. On DeepSeek-V4-Pro, **RepoTrace** reduces IT from 441.33M to 319.48M (-27.61%) while tripling MPR, since repository-level guidance prunes redundant exploration of the codebase. Considering that OT rises only moderately across all models and IT even decreases on the strongest performer, we regard the overall token overhead as acceptable. Overall, **RepoTrace** improves model performance on EvoCode-Bench with an acceptable efficiency cost.

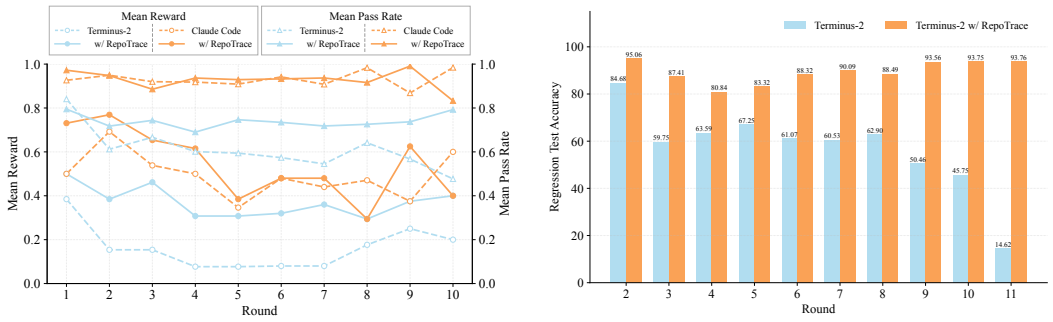
5.3 RQ2: To what extent does **RepoTrace** improve performance across different harnesses?

Table 2: RepoTrace across agent harnesses on EvoCode-Bench.

Agent	MPR ↑	FCR ↑	PTR ↑	HTR ↓	TCS ↑	IO ↓	OT ↓
Terminus-2	14.14	11.54	3.85	92.31	61.74	441.33	**12.51**
w/ RepoTrace	**36.39**	**34.62**	**15.38**	**61.54**	**73.63**	**319.48**	16.92
Claude Code	44.65	42.31	19.23	57.69	91.14	842.06	**7.99**
w/ RepoTrace	**54.87**	**46.15**	**26.92**	**50.00**	**93.02**	**675.36**	9.44

To answer RQ2, we evaluate **RepoTrace** on two harnesses (*i.e.,* Terminus-2 and Claude Code, both driven by DeepSeek-V4-Pro) to verify whether the effectiveness of **RepoTrace** generalizes across different agent harnesses. Table 2 reports the results.

RepoTrace improves performance across different harnesses. On the weaker Terminus-2, **RepoTrace** raises MPR from 14.14 to 36.39 (about $2.57\times$) and PTR from 3.85 to 15.38 (about $4\times$). On Claude Code, **RepoTrace** further improves MPR from 44.65 to 54.87 (+10.22 points), suggesting that the repository-level context extracted by **RepoTrace** complements the context extracted by the harness (see RQ4 and Appendix A). Token efficiency improves consistently as well when using DeepSeek-V4-Pro: IT decreases by 27.61% on Terminus-2 and by 19.80% on Claude Code, while OT rises only slightly on both harnesses. Overall, **RepoTrace** brings consistent improvement across harnesses at an acceptable efficiency cost.



(a) Mean Reward and Mean Pass Rate (b) Regression Test Accuracy

Figure 3: Results of DeepSeek-V4-Pro on EvoCode-Bench using Terminus-2 and Claude Code, with and without **RepoTrace**. **(a)** Mean reward (left axis) and mean pass rate (right axis) over 10 rounds: the average reward and the average fraction of test cases passed per round, each averaged over tasks. **(b)** Regression test accuracy in multi-round interactions: for each round $i \in \{2, \ldots, 11\}$, we report the mean pass rate over all test cases belonging to rounds $1, \ldots, i-1$, averaged over 26 tasks. Higher values indicate fewer regression errors.

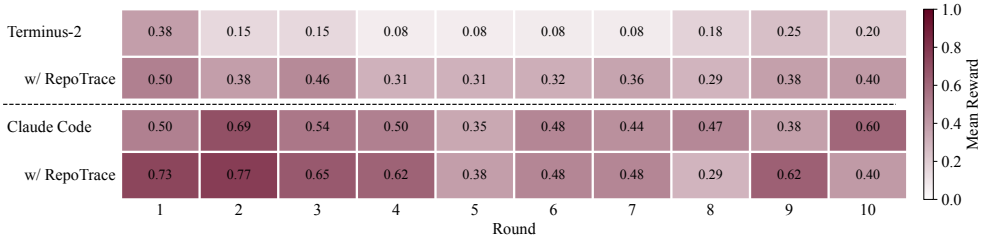


Figure 4: Heatmap of the per-round mean reward for each harness-**RepoTrace** combination.

5.4 RQ3: How does **RepoTrace**'s improvement vary across rounds?

To answer RQ3, we analyze the per-round behavior of DeepSeek-V4-Pro on EvoCode-Bench under both harnesses over 10 rounds. Figure 3 reports the per-round mean reward, mean pass ratio, and the regression errors on earlier rounds' test cases, and Figure 4 visualizes the per-round mean reward of each harness–**RepoTrace** combination.

RepoTrace mitigates error accumulation and regression across rounds. On Terminus-2, vanilla collapses after the first round, dropping from 0.38 in round 1 to 0.08 $\sim$ 0.15 in rounds 2 to 7. In contrast, **RepoTrace** maintains a mean reward between 0.29 and 0.50 and leads in all 10 rounds. The regression results further confirm this gap. Vanilla's pass rate on the test cases of earlier rounds degrades from 84.68 at round 2 to only 14.62 at round 11, while **RepoTrace** keeps this rate above 80 in every round. Averaged over rounds 2 to 11, **RepoTrace** improves this pass rate from 57.06 to 89.46 (+32.40 points, a 56.78% relative improvement), which means fewer regression errors. Overall, the benefit of **RepoTrace** persists as rounds advance, with a steady improvement in both harnesses.

5.5 RQ4: How Does **RepoTrace** Improve Requirement Tracing?

In RQ4, we use a case study to further examine how **RepoTrace** improves requirement tracing. JobForge is a nine-round software task from EvoCode-Bench. A code agent incrementally develops a Go command-line tool as new requirements arrive. Round 6 adds the `schedule` subcommand. Round 7 adds the `critical` subcommand for critical-path analysis while requiring existing functionality to remain correct. We compare Vanilla and **RepoTrace** trajectories produced with the same requirement chain, model, and execution environment. We independently replayed both trajectories and confirmed their outcomes. Figure 5 compares the code modifications under the two settings.

RepoTrace aligns code modifications focused on the current requirement and reduces unintended changes to existing functionality. ❶ In the Vanilla, the agent added `critical` to the shared command dispatcher but left the existing `schedule` branch without a call to

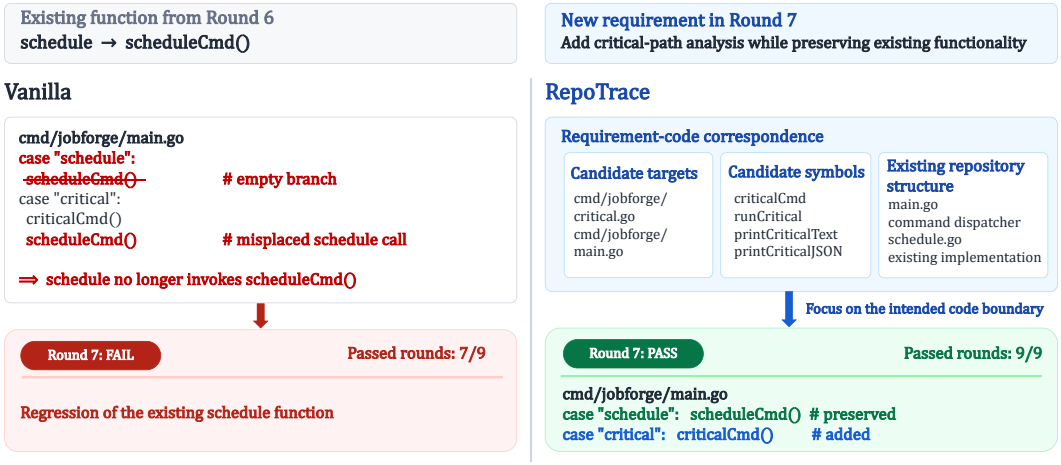


Figure 5: Case study of **RepoTrace** on JobForge. In the Vanilla setting, adding the `critical` subcommand breaks the existing `schedule` dispatch. In the **RepoTrace** setting, the requirement-code correspondence guides the agent to add an independent `critical` branch while preserving the `schedule` function, satisfying both the current and prior requirements.

`scheduleCmd()`. The command dispatcher routes each subcommand to its implementation. The missing call therefore broke the scheduling functionality introduced in Round 6. ❷ In the **RepoTrace**, the requirement-code correspondence identified `cmd/jobforge/critical.go` and `cmd/jobforge/main.go` as candidate primary modification targets, and `criticalCmd` and related critical-path functions as candidate symbols. The repository structure also showed the existing `schedule` implementation. Using the correspondence, the agent added a separate dispatch branch for `critical` and preserved the existing `schedule` → `scheduleCmd()` relation. Overall, **RepoTrace** combines requirement semantics with repository structure to identify primary modification targets and preserve existing functionality, reducing requirement focus misalignment.

6 Discussion

In this section, we discuss 1) the applicability of **RepoTrace** and 2) the role of requirements in agentic software engineering.

As a requirement-driven repository understanding method, **RepoTrace** can be extended to different harnesses through the requirement-code correspondence produced by **RepoTrace**. Both humans and code agents can modify the correspondence, allowing **RepoTrace** to work with different harnesses without directly modifying their execution process. Deeper integration (*e.g.,* using hooks) may provide more control but reduce the range of applicable **RepoTrace**. Supporting a wider range of harnesses while maintaining effective integration remains an open problem.

Requirements play a central role in agentic software engineering. Code agents need to translate requirements into code changes and retain previous requirements as the software evolves. As the interaction horizon grows, agents may get lost, focus on the most recent requirement, and miss requirements addressed in earlier rounds. Therefore, maintaining requirements throughout software development can help agents preserve their objectives while making new changes. More broadly, requirements provide a persistent source of objectives for agents to build on their previous experience, which is important for RSI.

7 Conclusion and Future Work

We present **RepoTrace**, a requirement-driven repository understanding harness that improves requirement tracing in multi-round agent interactions. **RepoTrace** helps code agents focus on relevant code and preserve existing functionality across interactions. In future work, we plan to: 1) explore how to manage and abstract requirement changes through structured representations, as with code, and 2) study how **RepoTrace** can help promote RSI.

AI USE STATEMENT

In preparing this manuscript, we use LLMs to assist with writing and coding. Specifically, LLMs improve clarity, grammar, and phrasing, ensuring the text is concise and readable. The use of LLMs does not influence the technical contributions or the interpretation of experimental findings. All content edited by LLMs was carefully checked by at least two authors.

REFERENCES

Anthropic. 2026 agentic coding trends report. Technical report, Anthropic, 2026. URL https://resources.anthropic.com/hubfs/2026%20Agentic%20Coding%20Trends%20Report.pdf.

Anthropic. Claude code. Product documentation, 2026. URL https://docs.anthropic.com/en/docs/claude-code/overview.

Hongyu Pei Breivold, Ivica Crnkovic, and Magnus Larsson. Software architecture evolution through evolvability analysis. *Journal of Systems and Software*, 85(11):2574–2592, 2012.

Nicholas Carlini, Keane Lucas, Evyatar Ben Asher, Newton Cheng, Hasnain Lakhani, David Forsythe, and Kyla Guru. Evaluating and mitigating the growing risk of llm-discovered 0-days. Technical report, Anthropic, 2026. URL https://www.anthropic.com/research/zero-days. Nicholas Carlini, Keane Lucas, and Evyatar Ben Asher contributed equally.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé 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, Joshua 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. *CoRR*, abs/2107.03374, 2021.

Mingyu Chen, Yakun Zhang, Zihao Xie, Yixing Luo, Jinrui Xu, Cuiyun Gao, Kaiqi Zhao, and Yunming Ye. Tracedev: A traceability-driven multi-agent framework for requirement-to-code development. *CoRR*, abs/2607.18886, 2026.

Zhaoling Chen, Robert Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor K. Prasanna, Arman Cohan, and Xingyao Wang. Locagent: Graph-guided LLM agents for code localization. In *ACL (1)*, pp. 8697–8727. Association for Computational Linguistics, 2025.

DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence. *CoRR*, abs/2606.19348, 2026.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. Swe-bench pro: Can AI agents solve long-horizon software engineering tasks? *CoRR*, abs/2509.16941, 2025.

Romina Etezadi, Sallam Abualhaija, Chetan Arora, and Lionel C. Briand. Llm-driven cost-effective requirements change impact analysis. *CoRR*, abs/2511.00262, 2025.

Dominik Fuchß, Tobias Hey, Jan Keim, Haoyu Liu, Niklas Ewald, Tobias Thirolf, and Anne Koziolek. Lissa: Toward generic traceability link recovery through retrieval- augmented generation. In *ICSE*, pp. 1396–1408. IEEE, 2025.

GLM. GLM-5: from vibe coding to agentic engineering. *CoRR*, abs/2602.15763, 2026.

Orlena C. Z. Gotel and Anthony C. W. Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of the 1st International Conference on Requirements Engineering (ICRE)*, pp. 94–101, 1994. doi: 10.1109/ICRE.1994.292398.

Fei Gu, Zi Liang, Jiahao Ma, and Hongzong Li. The matthew effect of ai programming assistants: A hidden bias in software evolution. In *International Conference on Learning Representations*, volume 2026, pp. 68445–68470, 2026.

Harbor Framework Team. Harbor: A framework for evaluating and optimizing agents and models in container environments. Open-source evaluation framework, 2026a. URL https://github.com/harbor-framework/harbor.

Harbor Framework Team. Terminus-2. Harbor reference agent documentation, 2026b. URL https://harborframework.com/docs/agents/terminus-2.

Tobias Hey, Dominik Fuchß, Jan Keim, and Anne Koziolek. Requirements traceability link recovery via retrieval-augmented generation. In *REFSQ*, volume 15588 of *Lecture Notes in Computer Science*, pp. 381–397. Springer, 2025.

Yonghui Huang, Lin Ma, Amjed Tahir, Qian Zhang, Liwen Xiao, and Lysa Xiao. Regression accumulation in multi-turn LLM programming conversations. *CoRR*, abs/2607.01855, 2026.

Philippe Laban, Hiroaki Hayashi, Yingbo Zhou, and Jennifer Neville. Llms get lost in multi-turn conversation. *CoRR*, abs/2505.06120, 2025.

Xunhao Lai, Weiqi Xu, Yufeng Yang, Qiaorui Chen, Yang Xu, Lunbin Zeng, Xiaolong Li, Haohai Sun, Haichao Zhu, Vito Zhang, Jinkai Hu, Jiayao Li, Rui Gao, Zekun Li, Songquan Zhu, Jingkai Zhou, and Pengyu Zhao. Minimax sparse attention, 2026. URL https://arxiv.org/abs/2606.13392.

Meir M Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

Bennet P Lientz, E. Burton Swanson, and Gail E Tompkins. Characteristics of application software maintenance. *Communications of the ACM*, 21(6):466–471, 1978.

Changzhi Liu, Yilun Liu, Sikuan Yan, Volker Tresp, and Yunpu Ma. Mendel gödel machine: Recursive self-improving coding agents via comparative evolution. *CoRR*, abs/2608.07645, 2026a.

Wei Liu, Ailun Yu, Daoguang Zan, Bo Shen, Wei Zhang, Haiyan Zhao, Zhi Jin, and Qianxiang Wang. Graphcoder: Enhancing repository-level code completion via coarse-to-fine retrieval based on code context graph. In *ASE*, pp. 570–581. ACM, 2024.

Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Fei Wang, Michael Qizhe Shieh, and Wenmeng Zhou. Codexgraph: Bridging large language models and code repositories via code graph databases. In *NAACL (Long Papers)*, pp. 142–160. Association for Computational Linguistics, 2025.

Zhongxin Liu, Zhonghao Jiang, Zhifan Ye, Haoye Wang, Jiakun Liu, and Xiaoxue Ren. Effective and efficient context retrieval via partial dependency graph for repository-level code generation. *CoRR*, abs/2608.01927, 2026b.

Jane Luo, Xin Zhang, Steven Liu, Jie Wu, Yiming Huang, Yangyu Huang, Chengyu Yin, Ying Xin, Jianfeng Liu, Yuefeng Zhan, Hao Sun, Qi Chen, Scarlett Li, and Mao Yang. RPG: A repository planning graph for unified and scalable codebase generation. *CoRR*, abs/2509.16198, 2025.

Muhammad Maaz, Liam DeVoe, Zac Hatfield-Dodds, and Nicholas Carlini. Finding bugs across the python ecosystem with claude and property-based testing. Technical report, Anthropic, 2026. URL https://www.anthropic.com/research/property-based-testing.

Mike A. Merrill, Alexander Glenn Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, Estefany Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis-Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Kumar Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *CoRR*, abs/2601.11868, 2026.

Moonshot AI. Kimi code, 2026. URL https://kimi.com/code. Accessed: 2026-09-07.

OpenAI. Introducing SWE-bench verified. https://openai.com/index/introducing-swe-bench-verified/, August 2024.

OpenAI. Codex. Product documentation, 2026. URL https://openai.com/codex.

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

Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking : Bringing order to the web. In *The Web Conference*, 1999. URL https://api.semanticscholar.org/CorpusID:1508503.

Qwen Team. Qwen3.7: The agent frontier, May 2026. URL https://qwen.ai/blog?id=qwen3.7.

Ruchit Rawal, Jeffrey Yang Fan Chiang, Chihao Shen, Jeffery Siyuan Tian, Aastha Mahajan, Tom Goldstein, and Yizheng Chen. Benchmarking correctness and security in multi-turn code generation. *CoRR*, abs/2510.13859, 2025.

Pratik Shah, Rajat Ghosh, Aryan Singhal, and Debojyoti Dutta. RANGER - repository-level agent for graph-enhanced retrieval. *CoRR*, abs/2509.25257, 2025.

Haiyang Shen, Xuanzhong Chen, Wendong Xu, Yun Ma, Liang Chen, and Kuan Li. Evocode-bench: Evaluating coding agents in multi-turn iterative interactions. *CoRR*, abs/2605.24110, 2026.

Theodore R. Sumers, Shunyu Yao, Karthik Narasimhan, and Thomas L. Griffiths. Cognitive architectures for language agents. *Trans. Mach. Learn. Res.*, 2024, 2024.

Ningzhi Tang, Chaoran Chen, Gelei Xu, Yiyu Shi, Yu Huang, Collin McMillan, Tao Dong, and Toby Jia-Jun Li. How coding agents fail their users: A large-scale analysis of developer-agent misalignment in 20,574 real-world sessions. *CoRR*, abs/2605.29442, 2026.

Peiding Wang, Li Zhang, Fang Liu, Lin Shi, Minxiao Li, Bo Shen, and An Fu. Codeif-bench: Evaluating instruction-following capabilities of large language models in interactive code generation. *CoRR*, abs/2503.22688, 2025.

Sizhe Wang, Zhengren Wang, Dongsheng Ma, Yongan Yu, Rui Ling, Zhiyu Li, Feiyu Xiong, and Wentao Zhang. Codeflowbench: A multi-turn, iterative benchmark for complex code generation. In *ACL (1)*, pp. 4369–4402. Association for Computational Linguistics, 2026.

Lu Yan, Xuan Chen, and Xiangyu Zhang. When the specification emerges: Benchmarking faithfulness loss in long-horizon coding agents. *CoRR*, abs/2603.17104, 2026.

Fu-Qing Yang. Thinking on the development of software engineering technology. *Journal of software*, 16(1):1–7, 2005.

Zexun Zhan, Shuzheng Gao, Ruida Hu, and Cuiyun Gao. Sr-eval: Evaluating llms on code generation under stepwise requirement refinement. *CoRR*, abs/2509.18808, 2025.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin gödel machine: open-ended evolution of self-improving agents. In *International Conference on Learning Representations*, volume 2026, pp. 104223–104294, 2026.

Tailin Zhou. Hierarchical self-improvement: A framework for task-specific evolvable agent harnesses. *CoRR*, abs/2608.08466, 2026.

A SPECIFIC EXAMPLES OF REPOTRACE

We use JobForge Round 7 to illustrate the three steps in Section 4: repository structure analysis, requirement semantics extraction, and agent context augmentation across rounds. JobForge is a Go command-line tool developed over multiple rounds. Round 6 introduces the `schedule` subcommand. In Round 7, the requirement r_7 asks for a `critical` subcommand that reports the longest dependency chain by job count and preserves all existing behavior. The following examples connect this requirement to the structural representation G_7, requirement entities Q_7, and requirement-code correspondence message M_7.

A.1 REPOSITORY STRUCTURE ANALYSIS

Following Section 4.1, the structural representation describes the repository before the current code modification. Figure 6(a) lists the 13 source files in V_7, including `main.go` and `schedule.go`. All listed paths share the prefix `cmd/jobforge/`. The structural change summary Δ_7 records `schedule.go` as an added file relative to the preceding snapshot. This entry captures a change observed before Round 7, rather than a file proposed for the current requirement.

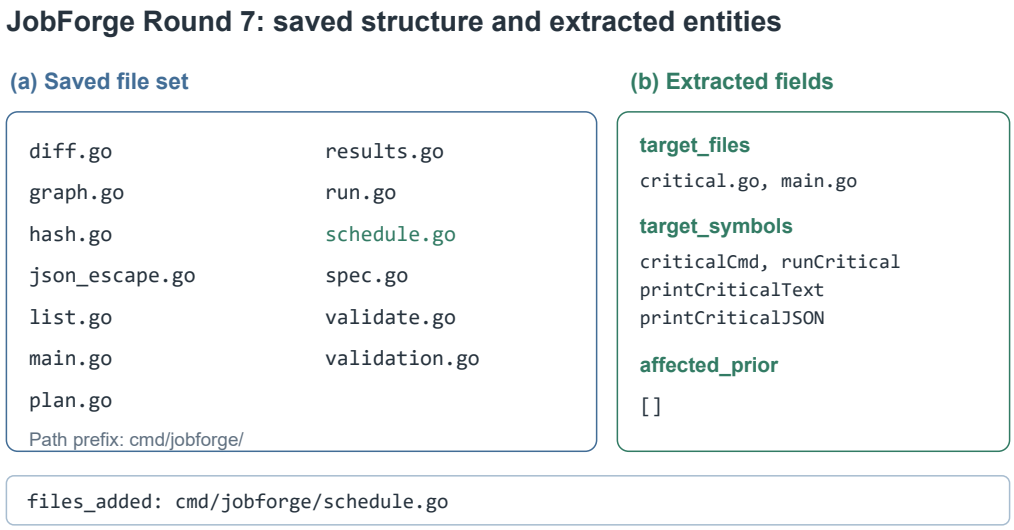


Figure 6: Repository structure and selected requirement entities for JobForge Round 7. Panel (a) shows the source-file set V_7, and panel (b) shows candidate targets and potentially affected prior code elements from Q_7. The bottom row shows the added-file entry in Δ_7. These records distinguish the existing `schedule.go` file from the candidate new file `critical.go`.

A.2 REQUIREMENT SEMANTICS EXTRACTION

To identify code elements associated with r_7, the extraction step uses the inputs defined in Section 4.2: the current requirement, repository structure, structural changes, and prior requirement records. Figure 7 shows how the reference prompt organizes these inputs. The file lists and symbol-to-file associations summarize G_t; {`instruction`} supplies r_t; and {`prev_block`} supplies $H_{<t}$ and Δ_t when prior records exist. The requested JSON fields correspond to the candidate target files and symbols, behavior constraints, potentially affected prior code elements, and summary in Q_t.

For JobForge Round 7, the saved extraction record identifies `critical.go` and `main.go` as candidate target files, together with symbols such as `criticalCmd` (Figure 6(b)). The behavior constraints specify the critical-path output and require all existing subcommands to remain unchanged. The `affected_prior` field is empty: no specific prior code elements are listed in that field. The preservation requirement is therefore represented by a behavior constraint, rather than by an explicit

list of affected prior code elements. These fields provide the requirement-specific information used in the next step.

Requirement semantics extraction prompt

Reference template from the available implementation.

```
You are analyzing one round of a multi-round software evolution task.

Below is the current repository structure (ranked by structural importance),
the accumulated requirement-code trace from previous rounds, and the NEW user
requirement for this round.

## Repository files (top {n_top} by structural weight, of {n_files} total)
{top_files}

## Full file inventory (truncated)
{all_files}

## Key symbols (name -> files, truncated)
{symbols}
{prev_block}
## New requirement for this round (round {round_no})
{instruction}

Extract the requirement entities FOR THIS ROUND. Use the previous-round trace
to tell apart what is ALREADY implemented (prior rounds' targets — usually do
NOT re-list them unless the new requirement modifies them) from what this
round newly demands. Return ONLY a JSON object (no prose):
{
  "target_files": ["repo-relative paths the requirement directly asks to
create/modify"],
  "target_symbols": ["functions/classes/types the requirement names or clearly
implies"],
  "behavior_constraints": ["observable behaviors that must hold after this
round"],
  "affected_prior": ["prior-round components whose behavior this round may break
or change"],
  "summary": "one sentence: what this round demands"
}
Only list files/symbols plausibly present (or to be created) in THIS repo.
```

Expansion of {prev_block} when prior records exist

```
## Requirement-code trace from previous rounds (already implemented)
{rounds}

## Structural changes since last round
{delta}
```

Wording preserved; line wrapping and Python brace escaping adjusted for display.

Figure 7: Reference prompt for requirement semantics extraction in Section 4.2. The upper box organizes the extraction inputs and specifies the fields of Q_t. The lower box expands {prev_block} with prior requirement records and structural changes. The template wording is retained; line wrapping and Python brace escaping are adjusted for display.

A.3 Agent Context Augmentation Across Rounds

The correspondence message connects the candidate targets in Q_7 to the repository described by G_7, illustrating the role of M_t in Section 4.3. Figure 8 shows excerpts from the actual Round 7 message. Comparing the target paths with V_7 identifies `critical.go` as a candidate new file and `main.go` as an existing file. The message thus distinguishes creating the requested implementation from modifying existing code.

To retain the context of earlier requirements, the recorded message also includes the Round 6 scheduling requirement and its candidate targets, `schedule.go` and `main.go`. Together with the constraint that existing subcommands remain unchanged, this history reminds the agent to preserve scheduling behavior when adding `critical`. The verification guidance asks the agent to check that affected code still builds and preserves existing behavior. The complete message appears verbatim in the first user input of the Round 7 agent trajectory, alongside the current requirement.

This example shows how prior requirement records supplement the current requirement before code modification. As described in Section 4.3, RepoTrace retains graph snapshots and requirement records for subsequent rounds. These records preserve earlier analyses; they do not certify that the agent has completed a requirement or passed its tests.

Message actually supplied to the code agent

Round 7 | Selected verbatim excerpts | [...] marks omitted lines

```
## 0. Requirement trace so far (already-implemented rounds)
  round 6: Add a schedule subcommand that deterministically assigns jobs to
worker lanes and outputs a byte-exact text or JSON plan without executing them.
             targets: cmd/jobforge/schedule.go, cmd/jobforge/main.go
  NOTE: keep ALL previously-implemented behavior working; this round must not
regress rounds above.
```

```
## 1. Architecture snapshot — 13 source files, 0 dependency edges (root: /app)
Core components (highest structural weight / PageRank):
[...]
  10. cmd/jobforge/schedule.go (fan-in=0, fan-out=0)
```

```
## 2. Requirement entities (extracted from this round's request, via llm)
Target files: cmd/jobforge/critical.go, cmd/jobforge/main.go
Target symbols: criticalCmd, runCritical, printCriticalText, printCriticalJSON
[...]
  - All existing subcommands unchanged
```

```
## 3. Requirement -> code mapping
  - file cmd/jobforge/critical.go :: NEW / not yet in repo
  - file cmd/jobforge/main.go :: cmd/jobforge/main.go (fan-in=0, fan-out=0)
[...]
```

```
## 4. Impact scope (dependents that must keep working)
Direct dependents of the mapped targets:
  (none — targets are currently leaf components)
```

```
## 5. Mandatory workflow for this round
[...]
  - Keep edits inside the target components + impact scope. Do NOT modify
unrelated components.
  - After editing, verify the impacted dependents still build and behave as
before (regression check).
```

Source: round_7_message.md; exact text also present in trajectory.json.

Figure 8: Excerpts from the requirement-code correspondence message M_7 supplied to the code agent. The message includes prior requirements, repository structure, candidate targets, existing/new-file mappings, and verification guidance. Ellipses mark omitted lines. The empty impact list reflects $E_7 = \emptyset$; the original phrase `leaf components` does not establish the absence of source-code dependencies.

B Setting for Requirement Focus Misalignment Analysis

We systematically analyze 2,656 agent trajectories from EvoCode-Bench. Following the terminology used in the main text, each agent trajectory denotes the interaction trace associated with one model–task round. These trajectories cover 26 multi-round software evolution tasks and 12 code models.

For each trajectory at round r, we define A_r as the set of source-code paths modified by the agent and O_r as the set modified by the benchmark-provided oracle implementation. We statically extract A_r from mutation operations recorded in the agent trajectory and extract O_r from the corresponding `solve.sh` and solution artifacts. File paths are normalized relative to the repository root and deduplicated within each round. We retain source files written in Go, Python, JavaScript/TypeScript, Java, C/C++, Rust, Ruby, PHP, and shell. Tests, third-party dependencies, build outputs, caches, temporary files, and common documentation files are excluded using fixed path-based rules.

Before computing the aggregate metric, we apply a conservative parsing-quality filter. The static parser resolves concrete file paths for 19,431 of the 19,926 detected mutation-related command instances, corresponding to a parsing coverage of 97.52%. We exclude any trajectory containing an unresolved mutation operation because its extracted modification set may be incomplete. This excludes 143 trajectories. One additional trajectory is excluded because its trajectory artifact is unavailable, leaving 2,512 trajectories with reliably extracted path sets.

The main analysis focuses on failed trajectories and excludes Round 1, which often involves repository initialization and has a qualitatively different modification scope from subsequent evolution rounds. This selection results in 1,846 trajectories. We measure file-level deviation from the oracle modification scope as

$$\mathrm{OffOracle} = \frac{\sum_r |A_r \setminus O_r|}{\sum_r |A_r|}. \tag{4}$$

Under this setting, 2,132 of the 3,865 agent-modified source-file instances fall outside the oracle modification scope, yielding an off-oracle ratio of 55.16%.

This metric measures disagreement with the modification scope of the benchmark reference implementation; it does not assume that the oracle provides the only valid implementation. The analysis is path-based and does not distinguish content-equivalent, whitespace-only, or comment-only writes. It also cannot verify whether every statically detected command successfully modified the workspace. We therefore report 55.16% as a corpus-level static estimate rather than a full-corpus replay ground truth. In a 30-trajectory calibration, the static and replay estimates under the same failed-trajectory, non-Round-1 setting are 32.26% and 34.29%, respectively.

C HTML Report of Repository Changes Across Rounds

We provide an HTML report of the JobForge trajectory in the Supplementary Material. The report organizes repository snapshots by round and displays candidate targets, extracted symbols, potentially affected prior code elements, and added files alongside file and function-call diagrams. Readers can browse the report in a web browser. Code agents equipped with file or browser tools can inspect the same HTML artifact. The supplied report is a static document that supports inspection of the recorded changes.

Figure 9 shows excerpts from Rounds 6–8. The file count increases from 12 to 13 and then 14. In Round 6, `schedule.go` appears as a candidate target. In Round 7, `schedule.go` appears in the added-file record, and `critical.go` becomes a candidate target. In Round 8, `critical.go` appears in the added-file record. These entries distinguish the targets of a current requirement from file additions observed in a subsequent snapshot. See the Supplementary Material for the complete nine-round report.

Round 6 PASS 12 files · 73 call edges src=llm	**targets:** cmd/jobforge/schedule.go, cmd/jobforge/main.go **symbols:** scheduleCmd, runSchedule **affected_prior:** cmd/jobforge/main.go (subcommand registration and flag setup) **files_added:** cmd/jobforge/diff.go

(a) Round 6: 12 files; `schedule.go` is a candidate target.

Round 7 PASS 13 files · 83 call edges src=llm	**targets:** cmd/jobforge/critical.go, cmd/jobforge/main.go **symbols:** criticalCmd, runCritical, printCriticalText, printCriticalJSON **affected_prior:** (none) **files_added:** cmd/jobforge/schedule.go

(b) Round 7: 13 files; `schedule.go` appears as an added file.

Round 8 PASS 14 files · 92 call edges src=llm	**targets:** cmd/jobforge/plan.go, cmd/jobforge/ancestors.go, cmd/jobforge/main.go **symbols:** PrintPlanText, ancestorsCmd, runAncestors, printAncestorsText, printAncestorsJSON **affected_prior:** cmd/jobforge/plan.go (PrintPlanText) **files_added:** cmd/jobforge/critical.go

(c) Round 8: 14 files; `critical.go` appears as an added file.

Figure 9: Cropped screenshots of three consecutive records in the supplementary HTML report. The report combines extraction fields with function-call graphs re-collected from a reconstructed agent workspace. Its call-edge counts therefore describe the supplementary reconstruction, not the saved file-relation sets in Appendix A.1.

Figures 10 and 11 show corresponding file-level details from the reconstructed function-call diagrams. The outer boxes group functions by file. Solid arrows denote cross-file calls, and dashed arrows denote calls within a file. The screenshots retain the original labels and show local regions of the larger diagrams; some cross-file arrows continue beyond the crop boundaries.

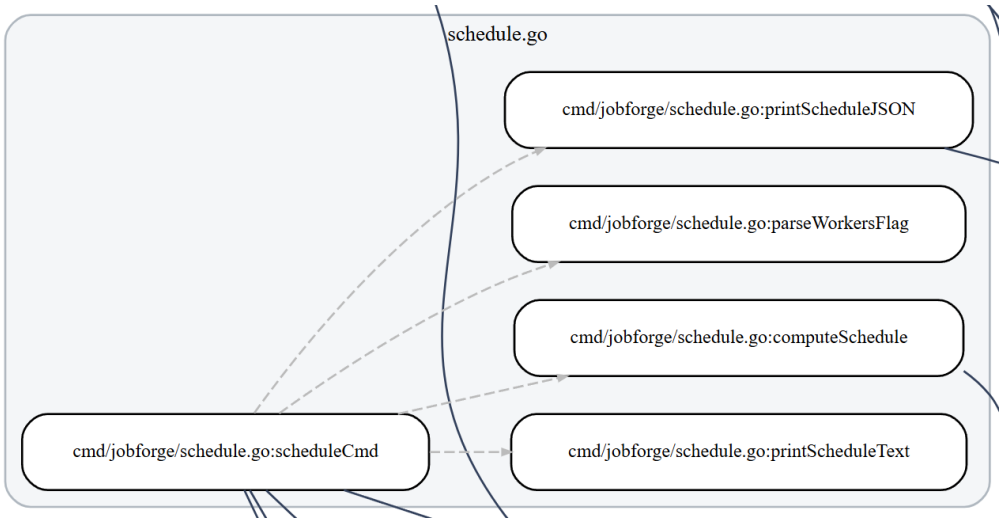


Figure 10: The `schedule.go` region of the Round 7 HTML diagram. The screenshot shows `scheduleCmd` and functions for parsing worker settings, computing the schedule, and formatting output. This file corresponds to the added-file entry in Figure 9(b).

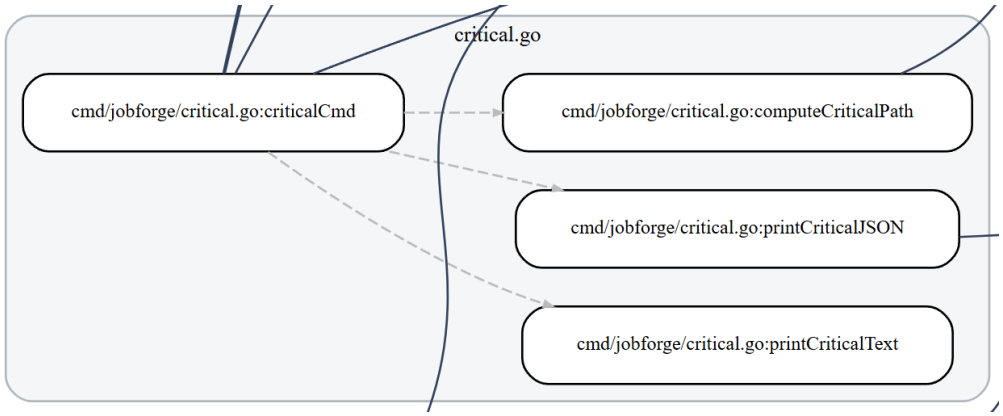


Figure 11: The `critical.go` region of the Round 8 HTML diagram. This file corresponds to the added-file entry in Figure 9(c). The full diagram, including connections to other files, is available in the supplementary HTML report.

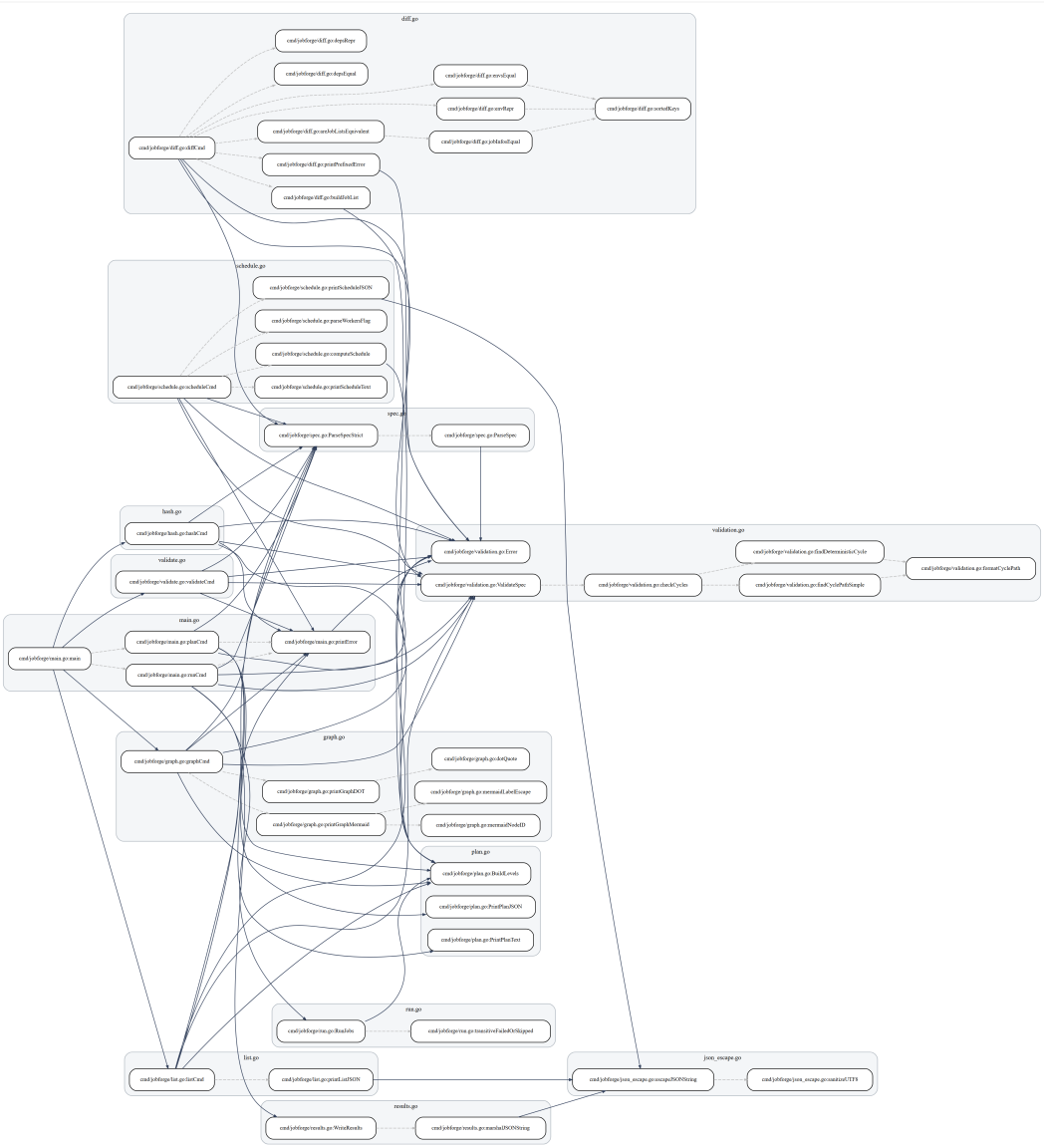


Figure 12: The overall repository structure and function-call graph for Round 7 in the supplementary HTML report.