GITHARNESS: GIT INIT YOUR HARNESS WORKING MEMORY FOR PERPETUAL USER REQUIREMENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM-based agents increasingly collaborate with users on long-horizon tasks, accumulating evidence, code, and drafts through extensive search, reasoning, and execution. As users inspect these results, they may supply missing information (***requirement completion***), introduce new requirements (***requirement elicitation***), or revise existing ones (***requirement shift***). These changes often affect only part of the accumulated work, yet agents may carry forward obsolete information or turn local revisions into global rewrites. Existing approaches clarify current intent without determining how prior work should change, or reuse execution histories under a fixed objective. We address this gap by formulating perpetual-requirement collaboration as joint requirement tracking and local update. We introduce **GITHARNESS**, a pluggable **Git**-style framework that organizes requirement states and their corresponding **harness** work states into a branchable version history. A trainable Git Agent resolves requirement changes and selects a semantically compatible historical state. A unified version interface then restores that state and creates a new branch, enabling the underlying harness to exclude obsolete information, inherit compatible work, and focus execution on affected parts. The Git Agent is trained through interface-level black-box reinforcement learning, with downstream harnesses and task-execution models kept fixed. We also construct MTAGENTBENCH, a verifier-preserving benchmark covering mathematical reasoning, text-to-SQL, agentic search, software engineering, and research synthesis. Experiments demonstrate strong task performance alongside effective requirement tracking, preservation of valid work, and efficient execution.

1 INTRODUCTION

LLM-based agents (Bubeck et al., 2023) can now carry out long-horizon tasks such as software engineering, open-world search, and deep research (Guo et al., 2025; Jaech et al., 2024; Jimenez et al., 2024; Zhou et al., 2024; Xiong et al., 2026; Du et al., 2025). During execution, they gather evidence, develop plans, invoke tools, and produce intermediate results such as code, queries, and report drafts. Yet users rarely provide a complete and immutable task specification at the outset. As they inspect intermediate results, they may supply missing task information (***requirement completion***), introduce new requirements inspired by the results (***requirement elicitation***), or revise, replace, or withdraw existing requirements (***requirement shift***) (Tack et al., 2026; Subramonyam et al., 2024; Lee et al., 2022; Faltings et al., 2023).

Unlike conventional multi-turn chat (Lee et al., 2025), the context of long-horizon agent collaboration includes not just dialogue but also the results of **extensive search, reasoning, and execution**. Subsequent user feedback often targets only part of this accumulated work. An agent must **track active requirements and reuse valid context and results**. In practice, however, a request to fix a single bug may trigger changes to unrelated files across the workspace (Zhu et al., 2026), while a request to refine one visual detail may alter the entire image (Figure 1). A local revision can thus become a global rewrite, unnecessarily changing work that was already correct. This motivates our central research question: ***How can an agent harness adapt to changing user requirements while excluding obsolete information, preserving unaffected work, and updating only affected parts?***

Retaining the full history risks carrying forward obsolete requirements and results, whereas restarting from current requirements discards potentially reusable work. Existing methods address this

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.

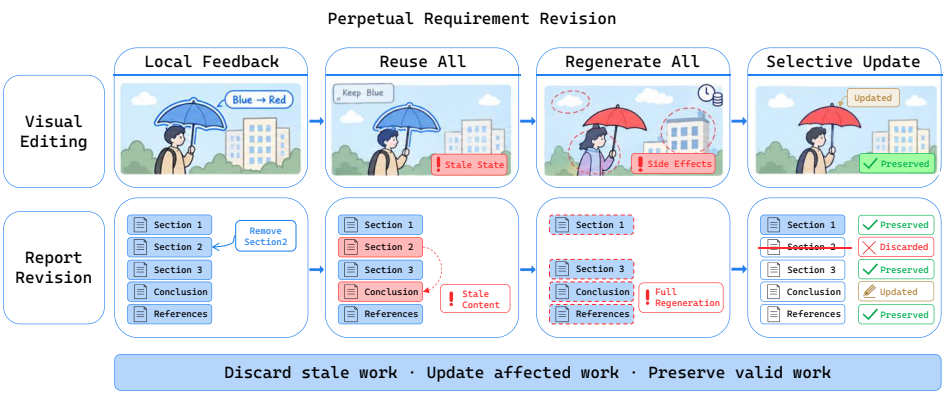


Figure 1: When user feedback changes the umbrella color or removes a report section, reusing all prior work leaves stale content, while regenerating everything may alter unaffected parts. Selective updating limits changes to the affected work.

trade-off only partially. ❶ Context retrieval, summarization, and intent reconstruction help clarify the current request (Dongre et al., 2025; Liu et al., 2026; Su et al., 2026), but do not directly determine how prior work should be reused, potentially requiring re-execution at substantial token and time cost. ❷ Trajectory organization, compression, and restoration support the reuse of intermediate work (Hu et al., 2025; Li et al., 2026; Hu et al., 2026; Zhuang et al., 2026), but typically assume a fixed task objective; when requirements change, superseded requirements and their associated results may continue to influence execution (Laban et al., 2026; Tack et al., 2026; Li et al., 2025). The missing link is an explicit connection between requirement changes and decisions about which prior work to preserve, discard, or recompute.

Addressing this gap requires two linked decisions: ***where to resume***—identifying the currently effective requirements and selecting a compatible historical state; and ***what to update***—determining which parts of the prior work must change and which can be preserved. **Version control** offers a natural abstraction for these decisions. Git preserves versions through commits, supports branching from past states, and records local changes as diffs. This suggests a way to *git init* an agent's working memory: organize requirements and their corresponding work as versioned states, and treat each new piece of feedback as a branch update from an appropriate historical state.

To this end, we introduce GITHARNESS, a pluggable Git-style version-tracking framework that supports dynamic user requirements across diverse agent harnesses. It couples each effective requirement state with its corresponding work state in a branchable version history. A trainable Git Agent identifies effective requirements, interprets their changes, and locates a semantically compatible historical state, translating multi-turn feedback into explicit version transitions. GITHARNESS then restores the selected work state and creates a new branch, allowing the underlying harness to exclude obsolete states, inherit compatible work, and focus further execution on the affected parts. A unified version interface maps these operations onto the native work states of different harnesses without redesigning their working memory or execution procedures. We train the Git Agent through interface-level black-box reinforcement learning, using the correctness of requirement resolution, change interpretation, and historical-state localization as feedback while keeping the downstream harnesses and task-execution models fixed. Our contributions are threefold:

- We formulate long-horizon agent collaboration under evolving requirements as a joint problem of requirement tracking and selective work reuse. The formulation covers requirement completion, elicitation, and shift, and connects changes in user requirements to decisions about where to resume execution and what to update.
- We propose GITHARNESS, a pluggable framework that couples requirement states and work states in a branchable version history. A trainable Git Agent interprets requirement changes and selects compatible historical states, while a unified version interface supports restoration and continued execution across different harnesses without retraining the underlying task executors.
- We construct MTAGENTBENCH, covering mathematical reasoning, text-to-SQL, agentic search, software engineering, and research synthesis under diverse requirement-change trajectories. The

dataset provides structured annotations of requirement states, version transitions, and task completion, together with an evaluation protocol for cross-task and compositional requirement changes.

2 Related Work

Dynamic user requirement tracking. A user request may be disclosed, extended, or revised over several turns, making the latest utterance insufficient on its own. Existing methods maintain user-side history by appending previous turns, retrieving relevant dialogue (Zhang et al., 2025), summarizing context, rewriting conversational queries (Yuan et al., 2025), or modeling relations such as follow-up, correction, expansion, and recall (Li et al., 2025). Intent-mismatch methods further separate intent mediation from task execution (Liu et al., 2026), while U-Fold constructs an intent-aware dialogue summary and a task-relevant tool log for evolving user intent (Su et al., 2026). Despite these advances, models remain anchored to early assumptions under progressive disclosure and struggle with intent revelation, revision, and redirection (Laban et al., 2026; Tack et al., 2026). These approaches clarify current intent but do **not determine how changes propagate to evidence, code, drafts, and other work produced under earlier requirements**.

Long-horizon agent work management. Long-horizon agents accumulate extensive traces and intermediate artifacts during planning and tool use. Memory methods organize, compress, or retrieve this execution-side history through stack operations, subgoal-based trajectories, active context editing, or state-adaptive recall (Jiang et al., 2025; Hu et al., 2025; Li et al., 2026; Hu et al., 2026). Recovery methods diagnose consequential failures or restore aligned agent and environment checkpoints (Qi et al., 2026; Ma et al., 2026; Zhuang et al., 2026). Related systems further introduce version-control abstractions: GCC manages context through commit, branch, merge, and retrieval (Wu et al., 2025), while GitOfThoughts stores reasoning trees as Git repositories (Shekar et al., 2026). These methods make prior execution easier to reuse, but select history according to the agent's current reasoning state, an execution failure, or exploration needs under an otherwise given task. They therefore **lack a requirement-driven mechanism for deciding which historical work state remains compatible when user feedback changes the task itself**. In contrast, GitHarness uses requirement transitions to locate compatible historical states and decide what work to preserve, discard, or recompute.

3 Problem Formulation

We study multi-turn tasks with evolving user requirements. At each turn t, the system maintains a *requirement state* $\boldsymbol{I}_{i,t}$ (current goal and constraints) and a *work state* $\boldsymbol{W}_{i,t}$ (execution context and accumulated artifacts). New feedback may change the goal and invalidate prior work. The central difficulty is that the agent cannot simply re-execute from scratch at each turn; it must identify what changed, retain valid prior work, and update only affected parts.

Requirement changes. User feedback typically modifies an existing requirement rather than defining a task from scratch. We identify three directions of change, each formalized as an edit $\Delta\boldsymbol{I}_{i,t}$ relative to a historical anchor state:

- **Completion**: adds missing details consistent with the current direction.
- **Elicitation**: introduces new task dimensions while preserving existing requirements.
- **Shift**: revises or withdraws prior requirements, invalidating part of the existing state.

The anchor need not be the most recent state—users may revisit earlier decisions or extend a historical branch. We instantiate nine fine-grained operators for data construction and evaluation; the method itself does not require change-type identification.

Task definition. Given the accepted history $\mathcal{H}_{i,t-1}$ and new feedback $q_{i,t}$, the system must (1) recover the full current requirement, (2) select a compatible historical base, and (3) update the work state locally:

$$(\widehat{\boldsymbol{Q}}_{i,t},\ \widehat{b}_{i,t},\ h_{i,t}) \ =\ f(q_{i,t},\ \mathcal{H}_{i,t-1})\,, \qquad \widehat{\boldsymbol{W}}_{i,t} \ =\ F_{d_i}\Big(\boldsymbol{W}_{i,\widehat{b}_{i,t}},\ \widehat{\boldsymbol{Q}}_{i,t},\ h_{i,t}\Big)\,. \tag{1}$$

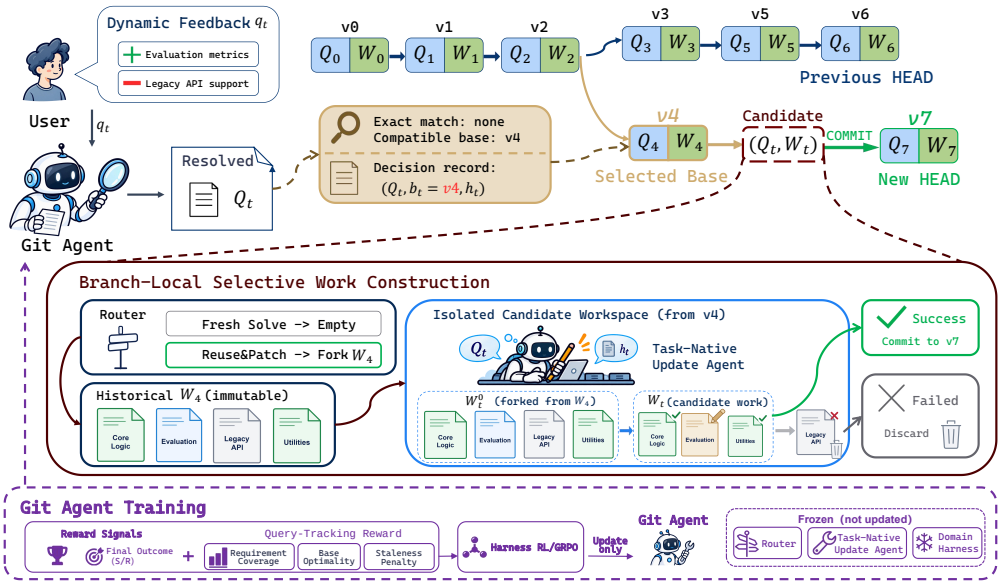


Figure 2: Framework of GITHARNESS.

Here $\widehat{\boldsymbol{Q}}_{i,t}$ is the resolved requirement, $\widehat{b}_{i,t}$ the selected base version, and $h_{i,t}$ an update hint for downstream execution. Rather than predicting which specific artifacts are affected, the system achieves locality through base selection and branch-local execution: resuming from a compatible prior state naturally confines edits to what the new requirement demands. For evaluation, each instance carries a final-task verifier and optional turn-level verifiers:

$$s_i^{\text{final}} = V_{d_i}^{\text{base}}\left(\widehat{\boldsymbol{Y}}_{i,T_i},\ \boldsymbol{I}_{i,T_i}\right), \qquad s_{i,t}^{\text{step}} = \widetilde{V}_{i,t}\left(\widehat{\boldsymbol{Y}}_{i,t},\ \boldsymbol{I}_{i,t},\ \Delta\boldsymbol{I}_{i,t}\right). \tag{2}$$

We additionally report execution cost.

4 METHOD

Flattening a growing interaction history into a single context obscures which prior states are reusable and makes selective rollback impossible. GITHARNESS addresses this by saving a *version* after each successful turn—a snapshot pairing the requirement with a recoverable checkpoint of the work state. Versions form a directed graph $\mathcal{G}_{<t}$; any past version can be restored on demand via its checkpoint $\mathcal{B}(v)$. This structure supports continuation from the current head, rollback to an earlier version, and branching when requirements diverge. When feedback q_t arrives, GITHARNESS processes via:

1. **Requirement resolution** (§4.1): A *Git Agent* determines the full current requirement $\boldsymbol{Q}_t$ and the best base version b_t.
2. **Branch execution** (§4.2): The system restores the selected base and an *update agent* carries out targeted modifications.

Separating these two concerns allows the Git Agent to focus purely on requirement understanding, while the update agent handles domain-specific execution. Formally:

$$(\boldsymbol{Q}_t,\ b_t,\ h_t) = \pi_\theta^G(q_t,\ \mathcal{G}_{<t}), \qquad \boldsymbol{W}_t = F(\boldsymbol{W}_{b_t},\ \boldsymbol{Q}_t,\ h_t). \tag{3}$$

Only normally terminated results are committed as new versions; failed executions leave the graph unchanged.

4.1 REQUIREMENT RESOLUTION

The Git Agent must jointly determine the full requirement and the best historical base—two interdependent decisions that require reasoning over the version graph. A single-step classifier is

insufficient: the agent may need to review what earlier versions achieved before deciding where to resume. It therefore runs a short ReAct loop with three actions:

- INSPECT: read a past version's requirement, gathering context without making a decision.
- COMMIT: declare the full current requirement and select a base version for runtime resolution.
- REUSE: select a version that already satisfies the requirement, skipping further execution.

At each reasoning step k:

$$\begin{aligned} a_k &\sim \pi_\theta^G(\cdot \mid q_t,\ \mathcal{G}_{<t},\ \eta_{<k}), \qquad a_k \in \{\textbf{INSPECT},\ \textbf{COMMIT},\ \textbf{REUSE}\}, \\ o_k &= \mathrm{Env}(\mathcal{G}_{<t},\ a_k). \end{aligned} \tag{4}$$

A successful COMMIT or REUSE terminates the loop and returns the handoff $\xi_t = (\boldsymbol{Q}_t, b_t, v_t, h_t)$, comprising the resolved requirement $\boldsymbol{Q}_t$, selected base b_t, target version v_t, and non-authoritative update hint h_t. COMMIT extends the main path when b_t is the current head and creates a branch otherwise—branching is thus a natural by-product of base selection, not a separate action.

4.2 BRANCH EXECUTION

The Git Agent has determined *what* to achieve and *where* to start; the remaining question is *how much work* is actually needed. Three execution modes cover the full spectrum:

$$z_t = \begin{cases} \textbf{EXACTREUSE}, & \mathcal{B}(v_t) \neq \perp, \\ \textbf{FRESHSOLVE}, & b_t = \varnothing \text{ or } \mathcal{B}(b_t) = \perp, \\ R_\psi(\boldsymbol{Q}_t,\ \boldsymbol{W}_{b_t}), & \text{otherwise.} \end{cases} \tag{5}$$

EXACTREUSE applies when a past version already satisfies the requirement—its checkpoint is returned directly with no execution. FRESHSOLVE starts from an empty workspace when no usable base exists. When a partially compatible base is available, the router R_ψ inspects a lightweight workspace manifest and chooses between FRESHSOLVE and REUSEANDPATCH; the latter forks the base checkpoint and applies targeted modifications, preserving unaffected work.

For the two modes that require execution, the update agent initializes and then updates a candidate state:

$$\boldsymbol{W}_t^{\,0} = \begin{cases} \mathrm{Empty}(), & z_t = \textbf{FRESHSOLVE}, \\ \mathrm{Fork}(\boldsymbol{W}_{b_t}), & z_t = \textbf{REUSEANDPATCH}, \end{cases} \qquad \widetilde{\boldsymbol{W}}_t = \mathrm{Execute}\big(\boldsymbol{W}_t^{\,0},\ \boldsymbol{Q}_t,\ h_t\big). \tag{6}$$

On normal termination the candidate is committed as a new version in the graph; on failure, the graph remains unchanged. Historical checkpoints are immutable, ensuring that failed executions never corrupt prior states.

4.3 TRAINING WITH HARNESS RL

Downstream quality depends on the Git Agent's requirement and base selection: even a perfect update agent cannot recover from errors in either. We therefore train only π_θ^G via reinforcement learning, treating the update agent and domain harness as a fixed environment. We adopt Harness RL (Jiang et al., 2026) to extract trainable segments from structured rollouts. For rollout capture, a single rollout involves interleaved calls to the Git Agent and the update agent. Every Git Agent call is recorded as an interface call record $\xi_n = (X_n, Y_n, \boldsymbol{\ell}_n^{\mathbf{roll}})$—tokenized input, sampled output, and per-token rollout log-probabilities. Between calls, the harness dispatches execution to the update agent and assembles resulting observations into the Git Agent's next context. Only Git Agent calls are included in the training mask; all other calls contribute context but receive no gradient.

Reward design. We combine a sparse outcome reward with a dense query-tracking reward, both attributed exclusively to Git Agent decisions. The outcome reward checks final-artifact correctness:

$$R_e^{\mathbf{out}} = V_{d_i}^{\mathbf{base}}\Big(\widehat{\boldsymbol{Y}}_{i,T_i},\ \boldsymbol{I}_{i,T_i}\Big). \tag{7}$$

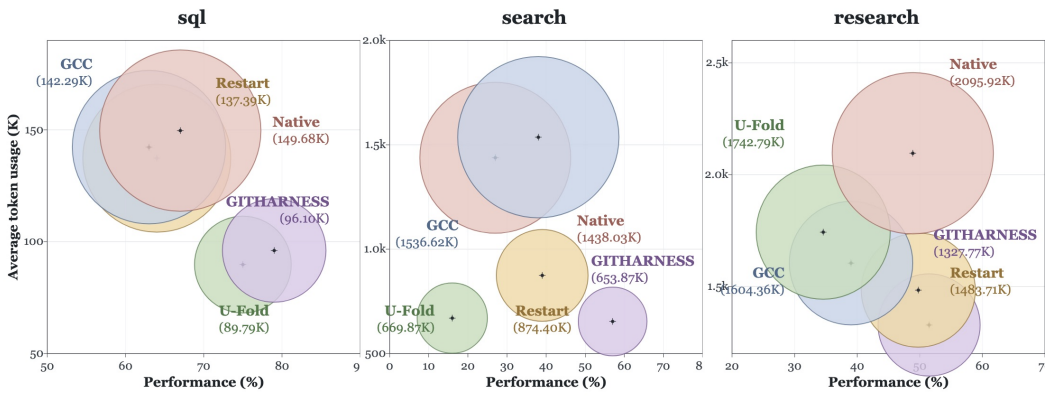


Figure 3: Performance versus token usage under StackPlanner with DeepSeek-V4-Flash. Bubble area denotes estimated API cost; lower right is better.

Because this signal cannot distinguish which turns contributed to success or failure, we introduce a turn-level query-tracking reward. At each turn t, an LLM-based judger J_ϕ scores whether the Git Agent faithfully tracks the user's evolving intent:

$$R_{e,t}^{\mathbf{track}} \;=\; J_\phi\Big(\widehat{\boldsymbol{Q}}_{i,t},\; \widehat{b}_{i,t},\; q_{i,t},\; \boldsymbol{I}_{i,t}\Big)\,. \tag{8}$$

The judger evaluates three dimensions: *requirement fidelity* (no omission or hallucination), *base optimality* (maximum compatible reuse), and *staleness exclusion* (withdrawn requirements correctly removed). These dimensions align with the evaluation metrics in Section 3, closing the loop between training signal and evaluation protocol.

Training objective. For each query we sample a group of K rollouts. Both reward components are group-normalized into advantages: $A_e^{\mathbf{out}}$ across the rollout group and $A_{n,t}^{\mathbf{track}}$ across same-type decisions. The per-token advantage for Git Agent positions combines them as

$$A_{n,t} \;=\; (1-\lambda_{\mathbf{track}})A_{e(n)}^{\mathbf{out}} \;+\; \lambda_{\mathbf{track}}\, A_{n,t}^{\mathbf{track}}\,. \tag{9}$$

We optimize π_θ^G with a rollout-balanced variant of GRPO (Shao et al., 2024):

$$\mathcal{L}_{\mathbf{GRPO}} \;=\; -\,\frac{1}{K}\sum_{e=1}^{K}\frac{1}{|\mathcal{I}_e|}\sum_{(n,t)\,\in\,\mathcal{I}_e}\min\Big(\rho_{n,t}\,A_{n,t},\;\; \mathrm{clip}(\rho_{n,t},\;1{-}\epsilon,\;1{+}\epsilon)\,A_{n,t}\Big), \tag{10}$$

where $\mathcal{I}_e$ collects Git Agent output-token positions in rollout e and $\rho_{n,t} = \exp\big(\ell_{n,t}(\theta) - \ell_{n,t}^{\mathbf{roll}}\big)$ is the importance ratio. The inner normalization by $|\mathcal{I}_e|$ ensures that rollouts with richer version-resolution histories do not dominate the gradient update.

5 Experiments

We evaluate GitHarness across five domains, two backbones, and three harnesses. We ask: **RQ1:** Does it improve performance and execution efficiency across models and harnesses? **RQ2:** Which components drive these gains? **RQ3:** Can interface-level RL improve the Git Agent, and how do training dynamics and reward balance affect adaptation? **RQ4:** Can non-parametric guidance distilled from prior inference improve performance without parameter updates? **RQ5:** Does GitHarness remain robust as interaction length and task concurrency increase?

5.1 Experimental Setup

Benchmarks. We extend five benchmarks into seven-turn interactions: GSM8K (Math; accuracy), BIRD (SQL; execution accuracy), BrowseComp-Plus (Search; answer accuracy), SWE-bench Verified (Code; resolved rate), and DeepResearch Bench (Research; RACE). Appendix C details the evaluation subsets, trajectory construction, and domain-specific protocols.

Table 1: Final-task performance across harnesses. Scores use each benchmark's native metric. Best results are shown in **bold**, and second-best results are underlined.

Harness	Cross-turn method	Qwen3-32B					DeepSeek-V4-Flash				
		Math	SQL	Search	Code	Research	Math	SQL	Search	Code	Research
TCRAG	Native	72.0	53.0	1.0	3.0	25.02	74.0	69.0	34.0	56.0	42.54
	+ Restart	73.0	64.0	2.0	5.0	16.67	88.0	66.0	41.0	61.0	43.45
	+ U-Fold	67.0	38.0	7.0	9.0	14.46	68.0	71.0	14.0	59.0	45.72
	+ GCC	71.0	63.0	6.0	11.0	15.66	55.0	53.0	42.0	53.0	32.37
	+ **GitHarness**	**89.0**	**70.0**	**8.0**	**19.0**	**38.45**	**93.0**	**80.0**	**56.0**	**62.0**	**46.65**
	+ **GitHarness-RL**	84.0	64.0	7.0	13.0	38.02	79.0	77.0	47.0	60.0	44.78
StackPlanner	Native	72.0	46.0	4.0	8.0	33.77	75.0	71.0	37.0	67.0	47.63
	+ Restart	80.0	66.0	3.0	16.0	19.50	76.0	72.0	45.0	68.0	46.83
	+ U-Fold	75.0	39.0	6.0	7.0	26.24	80.0	**81.0**	32.0	73.0	38.18
	+ GCC	62.0	61.0	11.0	10.0	21.04	61.0	56.0	46.0	66.0	41.45
	+ **GitHarness**	**86.0**	**72.0**	**15.0**	17.0	38.78	**94.0**	78.0	**58.0**	**74.0**	**49.96**
	+ **GitHarness-RL**	77.0	65.0	12.0	**18.0**	**39.20**	81.0	73.0	49.0	69.0	47.83
OpenHands	Native	71.0	58.0	2.0	7.0	20.22	76.0	67.0	27.0	64.0	48.88
	+ Restart	72.0	43.0	3.0	10.0	18.20	72.0	64.0	39.0	65.0	49.76
	+ U-Fold	47.0	22.0	2.0	6.0	8.99	71.0	75.0	16.0	59.0	34.57
	+ GCC	81.0	**77.0**	10.0	11.0	13.32	62.0	63.0	38.0	65.0	39.01
	+ **GitHarness**	85.0	73.0	10.0	**16.0**	**39.25**	**92.0**	**79.0**	**57.0**	**72.0**	**51.48**
	+ **GitHarness-RL**	**86.0**	63.0	**11.0**	15.0	37.92	87.0	78.0	48.0	69.0	46.17

Models and harnesses. For training-free GitHarness, Qwen3-32B (Yang et al., 2025) or DeepSeek-V4-Flash (DeepSeek-AI et al., 2026) serves as both the Git Agent and downstream task executor. For GitHarness-RL, the Git Agent is replaced by a LoRA-GRPO-trained Qwen3-8B, while downstream execution continues to use the model specified by the corresponding experimental setting. We evaluate these configurations with TCRAG (stack memory) (Jiang et al., 2025), StackPlanner (hierarchical multi-agent planning) (Zhang et al., 2026), and OpenHands (executable workspaces) (Wang et al., 2025). Appendix D provides additional details on the three harnesses.

Baselines. We compare Native, Restart, U-Fold (Su et al., 2026), and GCC (Wu et al., 2025), representing native continuation, clean-workspace regeneration, intent-aware context folding, and Git-style context management, respectively. Appendix E details their implementations.

5.2 Main Results

Table 1 compares methods within each fixed backbone–harness setting; absolute scores are not compared across harnesses because their tools and planning procedures differ.

Overall Effectiveness (RQ1). Across two backbones, three agent harnesses, and five task domains (Table 1), GitHarness outperforms *Native* in all 30 settings and matches or exceeds the strongest competing baseline in 28, including 27 strict wins and one tie. The gains are especially pronounced on Search: with DeepSeek-V4-Flash, GitHarness improves over the strongest baseline by **14.0** points under TCRAG and **18.0** points under OpenHands.

Consistency across Models and Harnesses (RQ1). Improvements over *Native* are observed with both backbones and all three harnesses. For example, with Qwen3-32B, GitHarness improves SQL by **26.0** points under StackPlanner and **15.0** points under OpenHands; with DeepSeek-V4-Flash, it improves Search by **22.0** points under TCRAG and **30.0** points under OpenHands.

Performance and Execution Efficiency (RQ1). Compared with *Native*, GitHarness reduces average token usage by **35.8%** on SQL, **54.5%** on Search, **73.6%** on Code, and **36.7%** on Research while improving performance in all four domains (Figure 3 and Table 10). It also achieves both the best performance and the lowest token usage among all compared methods on Search, Code, and

Table 2: Inference-time ablations under StackPlanner with DeepSeek-V4-Flash. Each task reports its native performance metric and average total token usage (thousands).

Component	Variant	Math		SQL		Search		Code		Research	
		Acc.↑	Tok.↓	EX↑	Tok.↓	EM↑	Tok.↓	Res.↑	Tok.↓	RACE↑	Tok.↓
Full GitHarness		94.0	60.0	78.0	96.1	58.0	653.9	74.0	1291.4	49.96	1327.8
Requirement Resolution	w/o Historical Version Selection	84.0	104.6	62.0	142.1	42.0	1892.6	44.0	4513.5	45.59	2299.6
	w/o Inspect	82.0	97.6	70.0	123.4	44.0	1583.5	42.0	4215.4	43.50	2412.3
	w/o Exact-State Reuse	88.0	107.8	76.0	130.3	54.0	963.3	63	2534.8	49.88	1360.6
Branch Execution	w/o ReuseAndPatch	92.0	65.4	75.0	104.3	53.0	862.4	69.0	1531.4	46.63	1597.0
	w/o FreshSolve	89.0	114.2	68.0	102.4	47.0	793.9	72.0	1032.5	42.56	1433.5

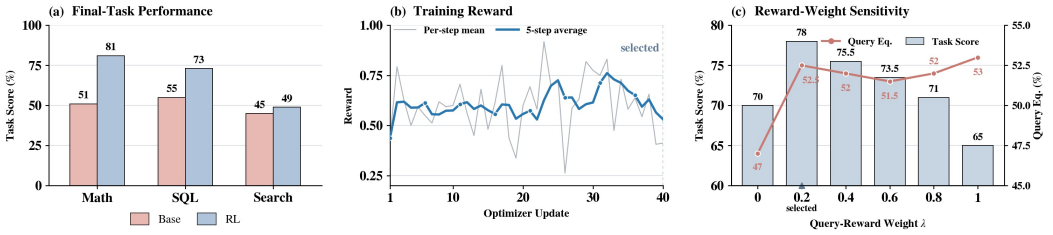


Figure 4: Harness RL adaptation with fixed downstream execution: (a) final-task performance on the held-out test set, (b) training reward, and (c) reward-weight sensitivity on the validation set.

Research. This lower cost is consistent with reusing compatible work and limiting re-execution after local requirement changes.

5.3 Ablation and Analysis

Component Ablation (RQ2). Table 2 shows that both requirement resolution and branch execution contribute to performance and efficiency. Removing *historical version selection* and falling back to *latest-state resolution*, which fixes the current head as the base instead of selecting an earlier version, reduces the Code resolved rate from 74.0 to 44.0 while increasing token usage from 1.29M to 4.51M; without Inspect, the resolved rate falls to 42.0 with 4.22M tokens. Disabling ReuseAndPatch lowers performance and increases token usage across all five domains, while disabling FreshSolve also degrades every task, including drops of **10.0** points on SQL and **11.0** points on Search. These results show the value of historical version selection and choosing the execution mode.

Harness RL Adaptation (RQ3). Table 1 reports the frozen test performance of the RL-adapted Git Agent, with the same held-out test comparison shown in Figure 4(a) under StackPlanner. With the downstream executor fixed, Harness RL improves the Qwen3-8B Git Agent from 51.0 to 81.0 on Math and from 55.0 to 73.0 on SQL, yielding gains of **30.0** and **18.0** points. Although the remaining gap to the training-free DeepSeek Git Agent may partly reflect the smaller 8B policy capacity, the RL-adapted configuration still outperforms the Qwen3-32B configuration on four of five StackPlanner domains.

Training Dynamics and Reward Balance (RQ3). Figure 4(b) reports the per-update training reward and its five-update moving average over 40 updates. Figure 4(c) compares query-reward weights on the 200-task validation set: compared with task-only training at $\lambda = 0$ (47.0 Query Eq. and 70.0 Task Score), $\lambda = 0.2$ improves both metrics. Increasing the weight to $\lambda = 1$ raises Query Eq. slightly to 53.0 but reduces Task Score to 65.0, showing that overemphasizing requirement tracking can hurt downstream completion. This result motivates the balanced reward used in the final policy.

Memory-Introduced Guidance (RQ4). In Table 3 and 11, we build a non-parametric guidance memory by distilling reusable strategies from multi-turn inference trajectories on separate source datasets and retrieve relevant guidance at test time. The memory is fixed and collected following

Table 3: Effect of skill injection under DeepSeek-V4-Flash. Parentheses report changes from the corresponding skill-free GitHarness.

	StackPlanner					OpenHands				
Method	**Math**	**SQL**	**Search**	**Code**	**Research**	**Math**	**SQL**	**Search**	**Code**	**Research**
GITHARNESS	94.0	78.0	**58.0**	74.0	49.96	92.0	79.0	57.0	72.0	51.48
GITHARNESS-Skills	**95.0** (+1.0)	**83.0** (+5.0)	**58.0** (0.0)	**75.0** (+1.0)	**52.03** (+2.07)	**94.0** (+2.0)	**81.0** (+2.0)	**58.0** (+1.0)	**74.0** (+2.0)	**53.60** (+2.12)

Table 4: Long-horizon performance under StackPlanner with DeepSeek-V4-Flash.

	Math					Research				
Method	$H = 10$	$H = 15$	$H = 20$	$H = 25$	$H = 30$	$H = 10$	$H = 15$	$H = 20$	$H = 25$	$H = 30$
Native	73.0	72.0	69.0	63.0	62.0	42.92	41.57	37.86	35.39	34.53
Restart	71.0	71.0	67.0	61.0	59.0	41.05	40.08	39.14	35.88	27.70
U-Fold	82.0	83.0	81.0	79.0	78.0	37.53	37.13	35.41	33.52	34.16
GITHARNESS	**92.0**	**90.0**	**87.0**	**85.0**	**84.0**	**49.53**	**49.27**	**48.71**	**47.52**	**47.32**

SkillRL Xia et al. (2026) when training, and contains no evaluation instances, reference answers, or task-specific evidence; Search, Code, and Research provide no trajectories for its construction. Memory guidance improves 13 of 15 settings, including a **5.0**-point gain on StackPlanner SQL and Research gains across all three harnesses, ranging from 2.07 to 4.27 points. These results show that experience accumulated during prior inference improves most of the evaluated settings, including held-out tasks, without parameter updates or test leakage.

Performance under Long-Horizon Interleaving (RQ5). Table 4 evaluates 10–30-turn sessions interleaving two to four tasks. GITHARNESS ranks first at every horizon on both Math and Research. At $H = 30$, it exceeds the strongest baseline by 6.0 points on Math and 12.79 RACE points on Research. From $H = 10$ to $H = 30$, its Research score decreases by only 2.21 points, compared with at least 3.37 points for the baselines, demonstrating robustness to increasing interaction length and task concurrency. Construction details are provided in Appendix C.

6 CONCLUSION, LIMITATIONS AND FUTURE WORK

GITHARNESS presents a Git-style version-tracking framework for long-horizon agent collaboration under dynamic user requirements. Instead of mixing evolving requirements and accumulated work in a flat interaction history, GITHARNESS binds each effective requirement to its corresponding harness state and organizes them as a branchable version history. Given users' new feedback, a Git Agent reconstructs the current requirement, locates a compatible historical state, and guides the underlying harness to preserve valid work, discard obsolete states, and update only the affected content. Across five task domains, two backbone models, and three agent harnesses, the training-free GITHARNESS configuration outperforms native cross-turn execution in all 30 settings and matches or exceeds the strongest competing baseline in 28 of 30 settings, including 14 of 15 settings under DeepSeek-V4-Flash. Moreover, after interface-level RL adaptation, the 8B Git Agent retains strong performance despite its smaller capacity, outperforming native cross-turn execution in 29 of 30 settings.

For future work, GITHARNESS depends on accurate requirement interpretation and historical-state selection, which can be challenging under ambiguous or conflicting feedback. Restoring a compatible state also does not guarantee that the underlying agent preserves all unaffected work. Future work will explore uncertainty-aware clarification and finer-grained dependency tracking, and evaluate the framework in longer, real-world interactions.

AI USE STATEMENT

Large language models were used for language polishing, literature discovery, synthetic trajectory construction, figure-design assistance, and AI-assisted programming. LLMs helped transform public benchmark tasks into multi-turn requirement trajectories and provided visual drafts, code suggestions, and debugging support. All generated data, figures, citations, and code were reviewed and verified by the authors, with benchmark verifiers and human inspection used to validate synthetic trajectories. The research questions, methodology, experimental design, analyses, and conclusions were determined by the authors.

ETHICS STATEMENT

All experiments use publicly available benchmarks, including GSM8K, BIRD, BrowseComp-Plus, SWE-bench Verified, and DeepResearch Bench, under their respective licenses and terms. The study involves no human or animal subjects and uses no private conversations or personally identifiable information. The multi-turn interactions are generated from public benchmark tasks using author-designed construction protocols and LLM assistance, rather than collected from real users. Software-engineering tasks are executed in isolated benchmark environments to limit unintended effects on external systems.

REPRODUCIBILITY STATEMENT

We will release the implementation of GITHARNESS, the constructed multi-turn trajectories, and the metadata required to reproduce their requirement transitions and verification. The appendix reports, prompts, model configurations, training settings, baseline implementations, and evaluation procedures. Evaluation scripts, LLM-judge configurations, and commands for reproducing the main results will be added to the anonymized repository at `https://anonymous.4open.science/r/GitHarness-7A2E/`. All experiments rely on publicly accessible benchmarks and model interfaces; no private data is required.

REFERENCES

Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott M. Lundberg, Harsha Nori, Hamid Palangi, Marco Túlio Ribeiro, and Yi Zhang. Sparks of artificial general intelligence: Early experiments with GPT-4. *CoRR*, abs/2303.12712, 2023. doi: 10.48550/ARXIV.2303.12712. URL `https://doi.org/10.48550/arXiv.2303.12712`.

Zijian Chen, Xueguang Ma, Shengyao Zhuang, Ping Nie, Kai Zou, Andrew Liu, Joshua Green, Kshama Patel, Ruoxi Meng, Mingyi Su, et al. Browsecomp-plus: A more fair and transparent evaluation benchmark of deep-research agent. *arXiv preprint arXiv:2508.06600*, 2025.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems, 2021. URL `https://arxiv.org/abs/2110.14168`.

DeepSeek-AI, Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chengyu Hou, Chenhao Xu, Chenze Shao, Chong Ruan, Conner Sun, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Donghao Li, Dongjie Ji, Erhang Li, Fang Wei, Fangyun Lin, Fangzhou Yuan, Feiyu Xia, Fucong Dai, Guangbo Hao, Guanting Chen, Guoai Cao, Guolai Meng, Guowei Li, Han Yu, Han Zhang, Hanwei Xu, Hao Li, Haofen Liang, Haoling Zhang, Haoming Luo, Haoran Wei, Haotian Yuan, Haowei Zhang, Haowen Luo, Haoyu Chen, Haozhe Ji, Hengqing Zhang, Honghui Ding, Hongxuan Tang, Huanqi Cao, Huazuo Gao, Hui Qu, Hui Zeng, J Yang, JQ Zhu, Jia Luo, Jia Song, Jia Yu, Jialiang Huang, Jialu Cai, Jian Liang, Jiangting Zhou, Jiasheng Ye, Jiashi Li, Jiaxin Xu, Jiewen Hu, Jieyu Yang, Jin Chen, Jin Yan, Jingchang Chen, Jingli Zhou, Jingting Xiang, Jingyang Yuan, Jingyuan Cheng, Jingzi Zhou, Jinhua Zhu, Jiping Yu, Joseph Sun, Jun Ran, Junguang Jiang, Junjie Qiu, Junlong Li, Junmin Zheng, Junxiao

Song, Kai Dong, Kaige Gao, Kang Guan, Kexing Zhou, Kezhao Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Wang, Leyi Xia, Li Zhang, Liang Zhao, Lihua Guo, Lingxiao Luo, Linwang Ma, Linyan Zhu, Litong Wang, Liyu Cai, Liyue Zhang, Longhao Chen, MS Di, MY Xu, Max Mei, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Mingxu Zhou, Minmin Han, Ning Wang, Panpan Huang, Panpan Wang, Peixin Cong, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qingyang Li, Qinyu Chen, Qiushi Du, Qiwei Jiang, Rui Tian, Ruifan Xu, Ruijie Lu, Ruiling Xu, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runqian Chen, Runqiu Yin, Runxin Xu, Ruomeng Shen, Ruoyu Zhang, Ruyi Chen, SH Liu, Shanghao Lu, Shangmian Sun, Shangyan Zhou, Shanhuang Chen, Shaofei Cai, Shaoheng Nie, Shaoqing Wu, Shaoyuan Chen, Shengding Hu, Shengyu Liu, Shiqiang Hu, Shirong Ma, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, Shuying Yu, Songyang Zhou, Tao Ni, Tao Yun, Tian Jin, Tian Pei, Tian Ye, Tianle Lin, Tianran Ji, Tianyi Cui, Tianyuan Yue, Tingting Yu, Tun Wang, W Zhang, WL Xiao, Wangding Zeng, Wei An, Weilin Zhao, Wen Liu, Wenfeng Liang, Wenjie Pang, Wenjing Luo, Wenjing Yao, Wenjun Gao, Wenkai Yang, Wenlve Huang, Wenqing Hou, Wentao Zhang, Wenting Ma, Xi Gao, Xiang He, Xiangwen Wang, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaokang Zhang, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingchen Liu, Xingkai Yu, Xingyou Li, Xinyu Yang, Xinyu Zhang, Xu Chen, Xuanyu Wang, Xuecheng Su, Xueyin Chen, Xuheng Lin, Xuwei Fu, YC Yan, YQ Wang, YW Ma, Yanfeng Luo, Yang Zhang, Yanhong Xu, Yanru Ma, Yanwen Huang, Yao Li, Yao Li, Yao Xu, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Qian, Yi Shao, Yi Yu, Yichao Zhang, Yifan Ding, Yifan Shi, Yijia Wu, Yiliang Xiong, Yiling Ma, Ying He, Ying Tang, Ying Zhou, Yingjia Luo, Yinmin Zhong, Yishi Piao, Yisong Wang, Yixiang Zhang, Yixiao Chen, Yixuan Tan, Yixuan Wei, Yiyang Ma, Yiyuan Liu, Yonglun Yang, Yongqiang Guo, Yongtong Wu, Yu Wu, YuKun Li, Yuan Cheng, Yuan Ou, Yuanfan Xu, Yuanhao Li, Yuduan Wang, Yuehan Yang, Yuer Xu, Yuhan Wu, Yuhao Meng, Yuheng Zou, Yukun Zha, Yunfan Xiong, Yupeng Chen, Yuping Lin, Yuqian Cao, Yuqian Wang, Yushun Zhang, Yuting Yan, Yutong Lin, Yuxian Gu, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuxuan Zhou, Yuyang Zhou, Yuzhen Huang, ZF Wu, Zehao Wang, Zehua Zhao, Zehui Ren, Zekai Zhang, Zhangli Sha, Zhe Fu, Zhe Ju, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zheren Gao, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhixian Huang, Zhixuan Chen, Zhiyu Wu, Zhizhou Ren, Zhongyu Wu, Zhuoshu Li, Zhuping Zhang, Zian Xu, Zihao Wang, Zihua Qu, Zihui Gu, Zijia Zhu, Zilin Li, Zipeng Zhang, Ziwei Xie, Ziyi Gao, Ziyi Wan, Zizheng Pan, and Zongqing Yao. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026. URL https://arxiv.org/abs/2606.19348.

Vardhan Dongre, Ryan A Rossi, Viet Dac Lai, David Seunghyun Yoon, Dilek Hakkani-Tür, and Trung Bui. Drift no more? context equilibria in multi-turn llm interactions. *arXiv preprint arXiv:2510.07777*, 2025.

Mingxuan Du, Benfeng Xu, Chiwei Zhu, Xiaorui Wang, and Zhendong Mao. Deepresearch bench: A comprehensive benchmark for deep research agents. *arXiv preprint arXiv:2506.11763*, 2025.

Felix Faltings, Michel Galley, Kianté Brantley, Baolin Peng, Weixin Cai, Yizhe Zhang, Jianfeng Gao, and William B Dolan. Interactive text generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 4450–4468, 2023.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Mengkang Hu, Tianxing Chen, Qiguang Chen, Yao Mu, Wenqi Shao, and Ping Luo. Hiagent: Hierarchical working memory management for solving long-horizon agent tasks with large language model. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 32779–32798, 2025.

Yuyang Hu, Hongjin Qian, Shuting Wang, Jiongnan Liu, Ziliang Zhao, Jiejun Tan, Zheng Liu, and Zhicheng Dou. Sam: State-adaptive memory for long-horizon reasoning agent. *arXiv preprint arXiv:2605.24468*, 2026.

Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. Openai o1 system card. *arXiv preprint arXiv:2412.16720*, 2024.

Xinke Jiang, Yue Fang, Rihong Qiu, Haoyu Zhang, Yongxin Xu, Hao Chen, Wentao Zhang, Ruizhe Zhang, Yuchen Fang, Xinyu Ma, Xu Chu, Junfeng Zhao, and Yasha Wang. TC-RAG: turing-complete rag's case study on medical LLM systems. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pp. 11400–11426. Association for Computational Linguistics, 2025. doi: 10.18653/V1/2025.ACL-LONG.558. URL https://doi.org/10.18653/v1/2025.acl-long.558.

Xinke Jiang, Zhixin Zhang, Zhibang Yang, Jiaran Gao, Rihong Qiu, Shijin Chen, Xu Chu, Junfeng Zhao, and Yasha Wang. Harness-rl: Black-box reinforcement learning with action-args decoupling for central-agent multi-agent harnesses. *arXiv preprint arXiv:2608.29641*, 2026.

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*, volume 2024, pp. 54107–54157, 2024.

Philippe Laban, Hiroaki Hayashi, Yingbo Zhou, and Jennifer Neville. Llms get lost in multi-turn conversation. In *International Conference on Learning Representations*, volume 2026, pp. 54738–54778, 2026.

Mina Lee, Megha Srivastava, Amelia Hardy, John Thickstun, Esin Durmus, Ashwin Paranjape, Ines Gerard-Ursin, Xiang Lisa Li, Faisal Ladhak, Frieda Rong, et al. Evaluating human-language model interaction. *arXiv preprint arXiv:2212.09746*, 2022.

Young-Jun Lee, Byung-Kwan Lee, Jianshu Zhang, Yechan Hwang, Byungsoo Ko, Han-Gyu Kim, Dongyu Yao, Xuankun Rong, Eojin Joo, Seung-Ho Han, Bowon Ko, and Ho-Jin Choi. Multiverse: A multi-turn conversation benchmark for evaluating large vision and language models, 2025. URL https://arxiv.org/abs/2510.16641.

Jinnan Li, Jinzhe Li, Yue Wang, Yi Chang, and Yuan Wu. Structflowbench: A structured flow benchmark for multi-turn instruction following. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 9322–9341, 2025.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36:42330–42357, 2023.

Mo Li, LH Xu, Qitai Tan, Long Ma, Hongyong Song, Ting Cao, and Yunxin Liu. Sculptor: Empowering llms with cognitive agency via active context management. In *International Conference on Learning Representations*, volume 2026, pp. 153411–153440, 2026.

Geng Liu, Fei Zhu, Rong Feng, Changyi Ma, Shiqi Wang, and Gaofeng Meng. Intent mismatch causes llms to get lost in multi-turn conversation, 2026. URL https://arxiv.org/abs/2602.07338.

Ming Ma, Jue Zhang, Fangkai Yang, Yu Kang, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. Dover: Intervention-driven auto debugging for llm multi-agent systems. In *International Conference on Learning Representations*, volume 2026, pp. 28059–28087, 2026.

Yunjia Qi, Zehua Yin, Xintong Shi, Hao Peng, Songyuanyi Lu, Yixian Liu, Richeng Xuan, Yuhong Liu, Zhichao Hu, Xiaozhi Wang, et al. Trajdebug: Tracing error lifecycle to identify critical failures in long-horizon agent trajectories. *arXiv preprint arXiv:2608.06346*, 2026.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Pavan C. Shekar, Abhishek H S, and Aswanth Krishnan. Gitofthoughts: Version-controlled reasoning and agent memory you can replay, diff, and merge. *arXiv preprint arXiv:2606.14470*, 2026.

Jin Su, Runnan Fang, Yeqiu Li, Xiaobin Wang, Shihao Cai, Pengjun Xie, Ningyu Zhang, and Fajie Yuan. U-fold: Dynamic intent-aware context folding for user-centric agents. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 18043–18057, 2026.

Hari Subramonyam, Roy Pea, Christopher Pondoc, Maneesh Agrawala, and Colleen Seifert. Bridging the gulf of envisioning: Cognitive challenges in prompt based interactions with llms. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, pp. 1–19, 2024.

Jihoon Tack, Philippe Laban, and Jennifer Neville. Llms get lost in evolving user intent. *arXiv preprint arXiv:2607.20734*, 2026.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, and et al. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=OJd3ayDDoF.

Junde Wu, Minhao Hu, Jiayuan Zhu, Jiazhen Pan, Yuyuan Liu, Min Xu, and Yueming Jin. Git context controller: Manage the context of llm-based agents like git. *arXiv preprint arXiv:2508.00031*, 2025.

Peng Xia, Jianwen Chen, Hanyang Wang, Jiaqi Liu, Kaide Zeng, Yu Wang, Siwei Han, Yiyang Zhou, Xujiang Zhao, Haifeng Chen, Zeyu Zheng, Cihang Xie, and Huaxiu Yao. Skillrl: Evolving agents via recursive skill-augmented reinforcement learning, 2026. URL https://arxiv.org/abs/2602.08234.

Zixun Xiong, Gaoyi Wu, Lingfeng Yao, Miao Pan, Xiaojiang Du, and Hao Wang. Webweaver: Breaking topology confidentiality in LLM multi-agent systems with stealthy context-based inference. *CoRR*, abs/2603.11132, 2026. doi: 10.48550/ARXIV.2603.11132. URL https://doi.org/10.48550/arXiv.2603.11132.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Yifei Yuan, Zahra Abbasiantaeb, Mohammad Aliannejadi, and Yang Deng. Query understanding in llm-based conversational information seeking. In *Proceedings of the 48th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 4098–4101, 2025.

Feiyuan Zhang, Dezhi Zhu, James Ming, Yilun Jin, Di Chai, Liu Yang, Han Tian, Zhaoxin Fan, and Kai Chen. Dh-rag: A dynamic historical context-powered retrieval-augmented generation method for multi-turn dialogue, 2025. URL https://arxiv.org/abs/2502.13847.

Ruizhe Zhang, Xinke Jiang, Zhibang Yang, Zhixin Zhang, Jiaran Gao, Yuzhen Xiao, Hongbin Lai, Xu Chu, Junfeng Zhao, and Yasha Wang. Stackplanner: A centralized hierarchical multi-agent system with task-experience memory management. *arXiv preprint arXiv:2601.05890*, 2026.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, volume 2024, pp. 15585–15606, 2024.

Tongyao Zhu, Wei Hern Lim, and Min-Yen Kan. When models edit too much: On the fidelity of minimal code edits, 2026. URL https://arxiv.org/abs/2609.04061.

Yu Zhuang, Kefei Chen, Yitong Duan, Shuxin Zheng, Jian Li, and Xu-Yao Zhang. Agentrewind: Recoverable execution for long-horizon llm agents. *arXiv preprint arXiv:2608.14380*, 2026.

Table 5: Requirement operators used to construct multi-turn interactions. Operator labels are hidden from evaluated agents.

Change type	Operator	Effect
Completion	REVEAL	Disclose previously unstated source-task information
	OUTPUTCONTROL	Revise format, structure, length, or presentation
	REFINE	Request additional explanation, evidence, or detail
Elicitation	CONSTRAIN	Add a newly formed executable constraint
	COMPARISON	Add an object, method, metric, or comparison dimension
	PIVOT	Introduce a related function while retaining compatible context
Shift	PARAMETERUPDATE	Replace an active parameter or constraint value
	CORRECT	Correct an active premise, goal, or requirement
	WITHDRAW	Deactivate an active requirement

A DATASET CONSTRUCTION

Final-anchor construction. Following the retrospective construction of Tack et al. (2026), we use the source task and its verifier as the final anchor:

$$\boldsymbol{I}_{i,T_i} = \boldsymbol{I}_i^{\mathrm{src}}, \qquad V_i = V_i^{\mathrm{src}}, \tag{11}$$

We then construct plausible preceding states that lead to this anchor. Single-turn and multi-turn evaluation therefore share the same final objective and native verifier, isolating the effect of evolving requirements without new answer annotations.

Requirement transition operators. We instantiate the three requirement-change directions in Section 3 with nine operators:

$$\begin{aligned} \mathcal{O}_{\mathrm{Completion}} &= \{\textsc{Reveal}, \textsc{OutputControl}, \textsc{Refine}\}, \\ \mathcal{O}_{\mathrm{Elicitation}} &= \{\textsc{Constrain}, \textsc{Comparison}, \textsc{Pivot}\}, \\ \mathcal{O}_{\mathrm{Shift}} &= \{\textsc{ParameterUpdate}, \textsc{Correct}, \textsc{Withdraw}\}. \end{aligned} \tag{12}$$

These hidden construction variables are distinct from the Git Agent's INSPECT, COMMIT, and REUSE actions and the update agent's domain tools.

Multi-turn interaction construction. Given the final anchor, we construct preceding requirement states and present them chronologically. At turn t, the constructor selects a historical anchor $b_{i,t}$, applies an operator $o_{i,t}$, and realizes the change as natural-language feedback $q_{i,t}$:

$$\Delta \boldsymbol{I}_{i,t} = o_{i,t}(\boldsymbol{I}_{i,b_{i,t}}), \qquad \boldsymbol{I}_{i,t} = \boldsymbol{I}_{i,b_{i,t}} \oplus \Delta \boldsymbol{I}_{i,t}. \tag{13}$$

Intermediate states may depart from the source task through a transient edit δ and later recover through a semantic compensating edit $\bar{\delta}$:

$$(\boldsymbol{I} \oplus \delta) \oplus \bar{\delta} \equiv \boldsymbol{I}. \tag{14}$$

For example, WITHDRAW can remove a temporary OUTPUTCONTROL, while PARAMETERUPDATE and CORRECT can restore parameters or premises. A trajectory is retained only if its terminal requirement is semantically equivalent to the source anchor. Construction labels, anchors, and reference states are used only for validation and reward computation and are hidden from evaluated agents.

B ADDITIONAL METHOD DETAILS

Inference procedure. Algorithm 1 summarizes requirement resolution, branch execution, and version commitment in GITHARNESS. Normal harness termination and runtime checks do not use the benchmark answer. The task verifier remains restricted to reward computation and offline evaluation.

Algorithm 1 GitHarness inference.

Require: Feedback q_t, version graph $\mathcal{G}_{<t}$, checkpoint binding $\mathcal{B}$, and domain harness Γ_d
Ensure: Updated work state $\boldsymbol{W}_t$ or FAILURE

$\eta \leftarrow \varnothing$ and $\xi_t \leftarrow \perp$
repeat
  $a_k \sim \pi_\theta^G(\cdot \mid q_t, \mathcal{G}_{<t}, \eta)$
  $(o_k, \xi_t) \leftarrow \mathrm{Env}_G(\mathcal{G}_{<t}, a_k)$
  $\eta \leftarrow \eta \oplus (a_k, o_k)$
until $\xi_t \neq \perp$
Unpack $\xi_t = (\boldsymbol{Q}_t, b_t, v_t, h_t)$
if $\mathcal{B}(v_t) \neq \perp$ **then**
  return $\mathrm{Restore}_d(\mathcal{B}(v_t))$
end if
if $b_t = \varnothing$ or $\mathcal{B}(b_t) = \perp$ **then**
  $z_t \leftarrow$ FRESHSOLVE
else
  $z_t \leftarrow R_\psi(\boldsymbol{Q}_t, \boldsymbol{W}_{b_t})$
end if
if $z_t =$ FRESHSOLVE **then**
  $\widetilde{\boldsymbol{W}}_t^0 \leftarrow \mathrm{Empty}_d()$
else
  $\widetilde{\boldsymbol{W}}_t^0 \leftarrow \mathrm{Fork}_d(\mathcal{B}(b_t))$
end if
$(s_t, \widetilde{\boldsymbol{W}}_t) \leftarrow \mathrm{Execute}_d(\widetilde{\boldsymbol{W}}_t^0, \boldsymbol{Q}_t, h_t)$
if $s_t \neq$ FINISHED **then**
  return FAILURE
end if
$c_t \leftarrow \mathrm{PersistCheckpoint}_d(\widetilde{\boldsymbol{W}}_t)$
$\mathcal{B}(v_t) \leftarrow c_t$
return $\widetilde{\boldsymbol{W}}_t$

Table 6: Semantic action classes available to the update agent.

Class	Generic operations	Purpose
Inspect	GLOB, GREP, READ	Inspect files or recovered work units
Task	Domain tools	Perform domain-specific task operations
Edit	WRITE, PATCH, DELETE	Modify only the isolated candidate state
Finish	FINISH	Return the result and request runtime finalization

Update agent. For execution modes that require new work, the update agent operates only on an isolated candidate state. Its domain-specific action space is partitioned as

$$\mathcal{A}_d^U = \mathcal{A}_d^{\mathrm{inspect}} \cup \mathcal{A}_d^{\mathrm{task}} \cup \mathcal{A}_d^{\mathrm{edit}} \cup \{\textsc{Finish}\}, \tag{15}$$

where inspect actions read recovered state, task actions invoke native domain tools, edit actions update candidate artifacts, and FINISH requests normal termination. At update step j,

$$u_{t,j} \sim \pi^U\Big(\cdot \mid \boldsymbol{Q}_t, h_t, z_t, \widetilde{\boldsymbol{W}}_{t,j}, \zeta_{t,<j}\Big), \qquad (\widetilde{\boldsymbol{W}}_{t,j+1}, o_{t,j}^U) = \mathrm{Env}_d(\widetilde{\boldsymbol{W}}_{t,j}, u_{t,j}), \tag{16}$$

where π^U is the fixed update-agent policy and $\zeta_{t,<j}$ is its within-execution action–observation history. Concrete tools, artifact representations, and stopping protocols remain domain specific. REUSEANDPATCH denotes continued execution from a fork rather than a deterministic file patch, while FRESHSOLVE and REUSEANDPATCH are runtime initialization decisions rather than update-agent editing actions.

Harness RL rollout extraction. An episode contains interleaved calls to the Git Agent, version runtime, update agent, and domain tools. For interface call n, we record $\xi_{e,n} = (X_{e,n}, Y_{e,n}, \boldsymbol{\ell}_{e,n}^{\mathrm{roll}})$,

where $X_{e,n}$ is the assembled context, $Y_{e,n}$ the sampled response, and $\ell^{\text{roll}}_{e,n}$ its rollout log-probabilities. The training mask is

$$m_{e,n,j} = \mathbb{I}[\xi_{e,n} \text{ is a Git-Agent call and token } j \in Y_{e,n}], \qquad \mathcal{I}_e = \{(n,j) : m_{e,n,j} = 1\}. \quad (17)$$

Only positions in $\mathcal{I}_e$ contribute to the policy update. User, system, runtime, tool-observation, and update-agent tokens remain in the rollout context but receive no gradient. Episode rewards are assigned to the corresponding Git-Agent segments before group normalization and the GRPO update defined in Section 4.3.

C Experiment Datasets

We evaluate GitHarness on five benchmarks spanning mathematical reasoning, text-to-SQL generation, web search, software engineering, and deep research. Only Math and SQL provide training trajectories for RL adaptation; Search, Code, and Research are held out entirely from RL training and serve as out-of-domain transfer evaluations. The Math and SQL candidate pools contain 400 tasks in total: 200 are used for RL training and 200 are reserved for validation. Table 7 summarizes the task-disjoint training, validation, and test splits.

❶ **GSM8K (Cobbe et al., 2021) (Math).** GSM8K contains 8.5K linguistically diverse grade-school math problems requiring multi-step arithmetic reasoning. From 200 candidate tasks, we use 100 for RL training and hold out 100 for validation; a separate 100-task test set is used for final evaluation. We evaluate the final answer using normalized exact-match accuracy, where extracted numerical answers are normalized before comparison with the gold answer.

❷ **BIRD (Li et al., 2023) (SQL).** BIRD is a large-scale, database-grounded text-to-SQL benchmark containing 12,751 question–query pairs across 95 databases and 37 domains. From 200 candidate tasks, we use 100 for RL training and hold out 100 for validation; a separate 100-task test set is used for final evaluation. We report execution accuracy by executing the predicted and gold SQL queries on the same SQLite database and comparing their returned results; gold SQL and execution results remain evaluator-only.

❸ **BrowseComp-Plus (Chen et al., 2025) (Search).** BrowseComp-Plus extends BrowseComp with a fixed, human-verified document corpus designed to provide controlled conditions for evaluating search agents. We use 100 queries and evaluate the final answers using normalized exact-match accuracy. Citation validity is additionally recorded as diagnostic information but is not used to determine the primary accuracy score.

❹ **SWE-bench Verified (Jimenez et al., 2024) (Code).** SWE-bench Verified contains 500 human-validated software engineering tasks derived from real GitHub issues. We construct a fixed subset of 100 tasks and evaluate the resulting patches in isolated environments using the official SWE-bench evaluation harness, with the resolved outcome as the success criterion.

❺ **DeepResearch Bench (Du et al., 2025) (Research).** DeepResearch Bench contains 100 PhD-level research tasks spanning 22 expert-curated domains, with 50 English and 50 Chinese queries. We use all 100 tasks and evaluate generated reports using RACE, which measures report quality against the benchmark's reference reports under its predefined evaluation criteria.

❻ **Multi-turn interaction construction.** For each source task, we construct a multi-turn interaction sequence using three controlled transition classes. Preserve retains the current goal and active requirements while revealing compatible information, changing presentation, or requesting further explanation. Extend keeps the current task active while adding a constraint, comparison dimension, or related task function. Revise corrects, replaces, withdraws, or restores an active requirement. By construction, the final effective query is semantically equivalent to the original source task, allowing the final artifact to be evaluated with the benchmark's original evaluator.

❼ **Interleaved long-horizon conversations.** To evaluate multi-requirement tracking and cross-task interference, we compose multiple task-disjoint seven-turn interaction sequences into a single

Table 7: Task-disjoint training, validation, and test subsets. Only the training trajectories are used for RL adaptation.

Dataset	RL Train	Validation	Test	Lang.
GSM8K	100	100	100	EN
BIRD	100	100	100	EN
BrowseComp-Plus	0	0	100	EN
SWE-bench Verified	0	0	100	EN
DeepResearch Bench	0	0	100	EN/ZH

conversation with $H \in \{10, 15, 20, 25, 30\}$ user turns. The conversation interleaves two to four active tasks, and each turn may update one task or jointly update several tasks. When needed to reach the target length, we insert continuation or progress-check requests (e.g., "continue working" or "report the current progress") that leave the active requirements unchanged. Hidden task states and transition labels remain unavailable to the agent. The final turn requests the outputs of all active tasks, which are verified separately using their benchmark-specific evaluators and then macro-averaged.

D Harness and Adapter Details

We evaluate all methods with three agent harnesses: TC-RAG, StackPlanner, and OpenHands. Each harness retains its native reasoning and tool-use procedure, while a lightweight adapter connects it to the corresponding benchmark interface.

❶ **TC-RAG.** TC-RAG Jiang et al. (2025) employs a memory-stack mechanism for adaptive retrieval, reasoning, and planning. Through Push and Pop operations, it manages intermediate states, controls the retrieval process, and mitigates the accumulation of erroneous knowledge.

❷ **StackPlanner.** StackPlanner Zhang et al. (2026) is a centralized hierarchical multi-agent framework that decouples high-level coordination from subtask execution. It employs active task-level memory control and structured experience memory to support long-horizon coordination and the reuse of prior coordination experience.

❸ **OpenHands.** OpenHands (Wang et al., 2025) is a general-purpose platform for developing AI agents that interact with their environment by writing code, operating through a command line, and browsing the web. It also provides sandboxed environments for safe code execution.

E Baseline Implementation Details

We compare GitHarness with four baselines that adopt different strategies for cross-turn context management: native state retention, context reconstruction, context compression, and versioned context management.

❶ **Native.** Native directly relies on the conversational state and workspace maintained by the underlying task harness, without introducing any additional cross-turn context-management module. Information from previous turns is therefore retained only through the harness's native state mechanism.

❷ **Restart.** Restart reconstructs the accumulated user requirements at each turn into a self-contained query and executes it in a fresh harness session with a clean workspace. Previous reasoning traces and intermediate artifacts are discarded, so each turn is solved independently based on the reconstructed requirement.

❸ **U-Fold.** U-Fold (Su et al., 2026) retains the full user–agent dialogue and tool-call history while constructing a compact working context at each turn. It maintains an intent-aware, evolving dialogue

Table 8: Domain-specific tools and persistent work products.

Domain	Model-visible tools and constraints	Persistent work products
Math	Deterministic calculator and scoped workspace read/write/patch operations	`answer.md` and calculation evidence
SQL	SQL generation from the public schema and BIRD evidence; no database access during inference	`answer.sql`
Search	Read-only BrowseComp retrieval returning top-5 passages with 512-token snippets	`answer.json` and runtime-recorded retrieval evidence
Research	Bounded read-only BoCha web search	`report.md`, report outline, and runtime-recorded retrieval evidence
Code	Repository inspection, editing, shell execution, and repository-local test execution in an isolated workspace; no access to hidden tests	Repository patch

summary together with a compact, task-relevant tool log, which are used to support the current interaction.

❹ **GCC.** GCC (Wu et al., 2025) organizes agent context as a version-controlled persistent file system. It provides four explicit operations—COMMIT, BRANCH, MERGE, and CONTEXT—to checkpoint progress, explore alternative reasoning paths, integrate branch states, and retrieve historical context at different levels of granularity.

F Implementation Details

Domain-specific execution environments. We instantiate the downstream update agent with the capabilities required by each benchmark while leaving requirement resolution unchanged. Table 8 summarizes the model-visible tools, persistent work products, and final verification for each domain. Within each backbone–harness block, all compared methods use the same downstream model, task inputs, model-visible tools, execution budget, and verifier. Across all domains, gold annotations, hidden evaluation resources, and verifier feedback remain evaluator-only; method-specific persistent state remains part of the system being evaluated.

Reinforcement-learning adaptation. We train only the Qwen3-8B Git Agent with LoRA GRPO. During RL training, the fixed downstream executor and query-equivalence judge use DeepSeek-V4-Flash, and all non-Git-Agent components remain frozen. Each optimizer update contains four prompt groups, with two Math and two SQL cases, and samples eight independent rollouts per group. Advantages are normalized only among rollouts generated from the same prompt. The terminal episode score is assigned to every trainable Git-Agent segment in that episode, while tokens produced by the runtime, downstream executor, tools, and verifiers are excluded from the loss. The selected checkpoint is trained for 40 updates, corresponding to 160 prompt groups and 1,280 sampled episodes. The reward weight $\lambda = 0.2$ is selected on the 200-task validation set described in Appendix C. Table 9 reports the complete training configuration.

Long-horizon finalization. When the final message requests outputs for several active tasks, the experiment scheduler extracts one task-local request for each referenced task and invokes the same GitHarness procedure independently on its version history and workspace. Each invocation produces one terminal COMMIT or REUSE, after which the scheduler aggregates the task outputs into one response. Thus, a global final message may trigger several task-local terminal actions without changing the single-handoff semantics of GitHarness.

G Additional Experiments

❶ **Efficiency and Cost Details.** Figures 5 and 6 complement the main-text efficiency analysis. The former reports the Math and Code token–performance trade-offs omitted from Figure 3 for readability, while the latter compares estimated API cost across all five domains. We aggregate all model

Table 9: LoRA-GRPO training configuration for the Git Agent.

Parameter	Value	Parameter	Value
Models and execution			
Trainable policy	Qwen3-8B Git Agent	Downstream executor model	DeepSeek-V4-Flash
Query-equivalence judge	DeepSeek-V4-Flash	Adaptation	LoRA
Git-Agent tool steps	4	Downstream tool steps	16
Git-Agent context / response limit	16,384 / 4,096 tokens	Downstream response limit	4,096 tokens
Sampling			
Updates	40	Checkpoint interval	Every update
Prompt groups/update	4 (2 Math + 2 SQL)	Rollouts/group	8
Sampled episodes	1,280	Temperature / top-p	0.8 / 1.0
Optimization			
LoRA rank / alpha	16 / 32	LoRA dropout	0
Target modules	Q/K/V/O and MLP projections	Precision	bfloat16
Optimizer	AdamW	Learning rate	2×10^{-6}
Adam betas / weight decay	$(0.9, 0.95)$ / 0	Schedule	Constant
GRPO clip range	0.1	Maximum gradient norm	1.0
Base-policy KL coefficient	0.01	Query-reward weight λ	0.2

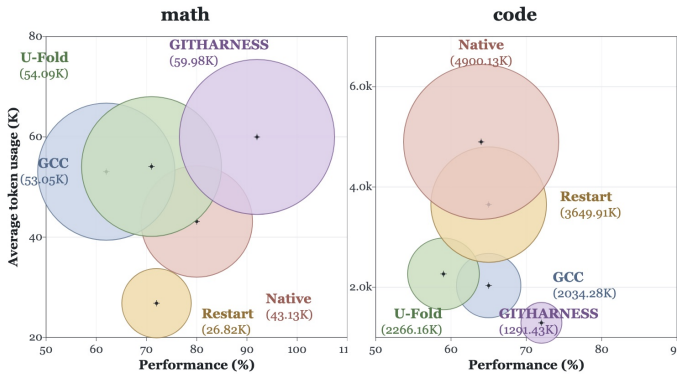


Figure 5: Performance versus token usage on Math and Code under StackPlanner with DeepSeek-V4-Flash. Bubble area denotes estimated API cost; lower right is better.

calls over each seven-turn interaction and average over completed samples. GITHARNESS achieves the lowest token usage and estimated cost on Search, Code, and Research; Math and SQL exhibit task-dependent trade-offs. Table 10 provides the corresponding numerical results.

❷ **Decision Behavior Analysis.** Figures 7 and 8 show that GITHARNESSdoes not collapse to a single resolution action or execution strategy. During requirement resolution, the Git Agent most frequently selects COMMIT in Math, SQL, Search, and Research, whereas Code more often reuses an existing requirement version; INSPECT is particularly common in Code and Research. During branch execution, FRESHSOLVE is most frequent in Math and SQL, REUSEANDPATCH dominates Research and is common in Search, and EXACTREUSE dominates Code. These domain-dependent

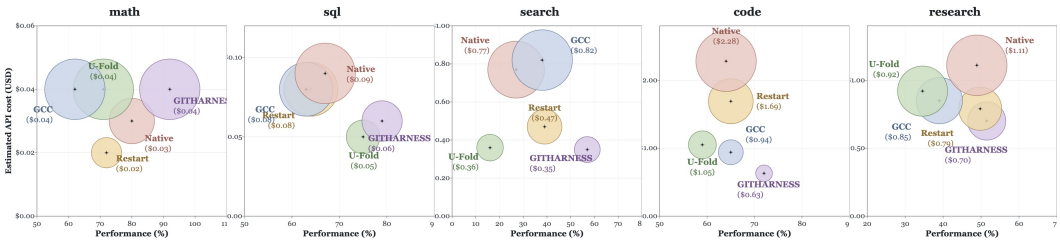


Figure 6: Performance versus estimated API cost across five domains under StackPlanner with DeepSeek-V4-Flash. Bubble area denotes average token usage; lower right is better.

Table 10: Detailed token usage (thousands) and API cost (USD) under StackPlanner with DeepSeek-V4-Flash.

	Math		SQL		Search		Code		Research	
Method	**Tokens**	**Cost**	**Tokens**	**Cost**	**Tokens**	**Cost**	**Tokens**	**Cost**	**Tokens**	**Cost**
Native	43.13	0.03	149.68	0.09	1,438.03	0.77	4,900.13	2.28	2,095.92	1.11
Restart	26.82	0.02	137.39	0.08	874.40	0.47	3,649.91	1.69	1,483.71	0.79
U-Fold	54.09	0.04	89.79	0.05	669.87	0.36	2,266.16	1.05	1,742.79	0.92
GCC	53.05	0.04	142.29	0.08	1,536.62	0.82	2,034.28	0.94	1,604.36	0.85
GITHARNESS	59.98	0.04	96.10	0.06	653.87	0.35	1,291.43	0.63	1,327.77	0.70

patterns show that GITHARNESS adapts both its historical-version selection and downstream execution strategy to the current update.

❸ Additional Skill-Injection Results. Table 11 complements the main analysis with results under TCRAG, a harness based on stack-structured task memory. Skill injection improves four of the five domains, including gains of 3.0 points on Math and 4.27 points on Research. Together with the StackPlanner and OpenHands results in Table 3, these results show that reusable version-decision experience transfers across harnesses with different state-management mechanisms.

H COMPUTATIONAL RESOURCES AND SOFTWARE ENVIRONMENT

Compute. Most inference experiments ran on a shared Ubuntu 22.04.5 LTS server with two Intel Xeon Platinum 8358 CPUs, 1007 GiB of RAM, and eight NVIDIA A800-SXM4-80GB GPUs. Harness-RL training was conducted separately on one NVIDIA H800 GPU.

Model serving. Qwen3-32B was served through an OpenAI-compatible vLLM 0.19.0 endpoint using two A800 GPUs, tensor parallelism of 2, bfloat16 precision, and a 61,440-token context limit. DeepSeek experiments used the official `deepseek-v4-flash` API with its default thinking configuration (enabled with high reasoning effort) and a 1M-token context window. All cross-turn methods within a backbone–harness–task cell used the same reasoning configuration, generation limits, and agent-step budget.

Task environments. BIRD predictions were executed against the benchmark's local SQLite databases and scored by execution accuracy. BrowseComp-Plus used its fixed 100,195-document corpus with Qwen3-Embedding-8B and FAISS retrieval on GPU; each search returned five passages truncated to 512 tokens. SWE-bench Verified instances ran in dedicated Docker 29.1.3 containers and were graded with the official harness. Research reports were evaluated with the official DeepResearch Bench pipeline, using GPT-5.5 for RACE.

Cost accounting. We report model-token cost per completed sample using peak cache-miss pricing from September 14, 2026: $0.44 per million input tokens and $1.32 per million output tokens.

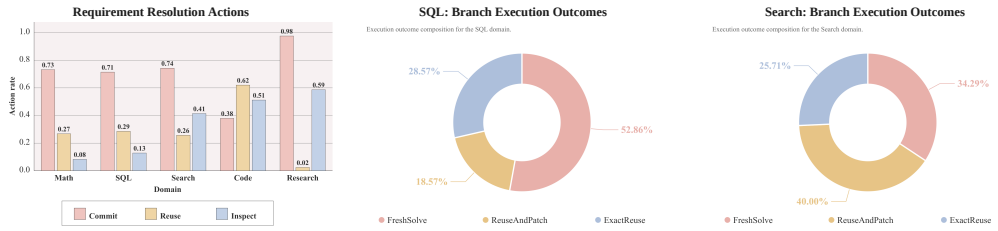


Figure 7: Requirement-resolution actions across all domains and branch-execution strategies for SQL and Search.

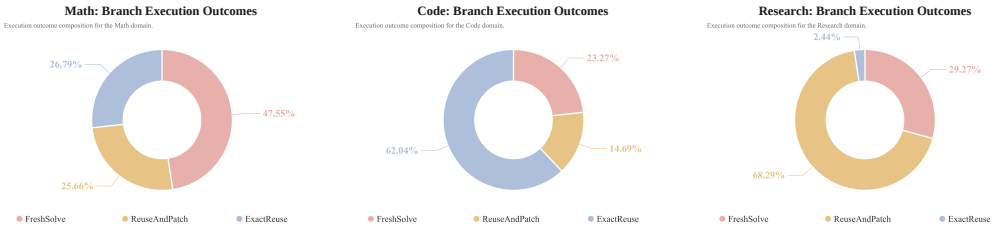


Figure 8: Branch-execution strategies for Math, Code, and Research.

I PROMPT TEMPLATES

This section presents the core prompts used by GITHARNESS. Because some prompts are lengthy, we omit repeated low-level instructions and examples for presentation; the released code provides the complete prompts used in all experiments.

I.1 REQUIREMENT RESOLUTION

The Git Agent uses the following prompt to convert each user turn into a complete resolved requirement and to choose among INSPECT, COMMIT, and REUSE. The box is an abridged presentation rather than a verbatim serialization; the implementation terms `state_forest` and `resolved_query` correspond to the paper's version graph and resolved requirement, respectively.

Git Agent Requirement-Resolution Prompt (Abridged)

You track the user's current requirement state over a version graph.
Use only the provided version-graph tools. Do not answer or execute the task. **Critical update procedure:**
1. Unless `current_request` explicitly withdraws or restores content, use `active_state.resolved_query` as the mandatory source. Edits are cumulative. Do not inspect or base an ordinary correction on an ancestor; reuse an ancestor only when its complete requirement exactly matches the desired requirement.
2. Treat entities, attributes, numbers, relationships, filters, projections, and outcomes as semantic slots. Replace every stale occurrence of a corrected slot, including evidence and question wording; preserve other slots and their required dependencies. Never leave conflicting values for one slot.
Steps 3–4, the context-schema explanation, repeated validation rules, formatting sanitization, and illustrative examples are omitted here for space.
5. Before COMMIT or REUSE, verify the latest requested outcome and output specification, the latest slot values and dependencies, all unmentioned active requirements, and the absence of withdrawn content. If any check fails, do not issue a terminal action.
Every `resolved_query` must be the complete, self-contained current requirement, never a delta such as "change X and keep everything else." Content fidelity has priority over brevity.

Table 11: Effect of skill injection with TCRAG under DeepSeek-V4-Flash.

Method	Math	SQL	Search	Code	Research
GITHARNESS	93.0	80.0	**56.0**	62.0	46.65
GITHARNESS-Skills	**96.0** (+3.0)	**81.0** (+1.0)	55.0 (-1.0)	**63.0** (+1.0)	**50.92** (+4.27)

Table 12: Values used for the domain-specific semantic-check placeholder.

Domain	Additional semantic check
Math	Preserve the mathematical goal, givens, variables, units, relationships, active constraints, and requested output.
SQL	Preserve the database question, requested columns or aggregation, filters, grouping, ordering, limits, comparisons, and aliases. Treat fixed schema and SQL transport wrappers as harness context.
Search	Preserve the information need, entities, disambiguating clues, time range, requested answer granularity, and evidence or citation requirements.
Research	Preserve the research question, scope, subjects, time and region bounds, requested synthesis, deliverable structure, evidence, and citation requirements. Do not reduce the task to short-answer search.
Code	Preserve the software issue goal, repository-scoped entities, affected symbols or files, reproduction and failure behavior, expected behavior, version constraints, requested change scope, and active testing or output requirements.

Silently distinguish the stable requirement core, persistent updates, temporary constraints, and presentation requirements such as precision, tables, or output format.
Apply this decision priority: explicit withdrawal or restoration, explicit goal change, persistent local edit, temporary addition, and exact no-op. A higher-priority instruction overrides lower-priority preservation rules. For an ordinary local edit, apply only the requested change to the active resolved requirement and preserve every unmentioned requirement-bearing clause. For an explicit goal change, rebuild the complete requirement around the new requested outcome while retaining accepted facts needed for that outcome and dropping content that served only the previous goal.
Detailed rules for comparison construction, restoration-scope edge cases, formatting sanitization, and illustrative examples are omitted here for space.
For explicit withdrawal or restoration, use the version index and parent links to identify the historical version immediately preceding the withdrawn content, inspect that candidate, and replay only later edits that remain active. If the desired requirement exactly matches an inspected version, call `reuse_resolved_state`; otherwise call `commit_resolved_query` with the appropriate base version. `commit_resolved_query` and `reuse_resolved_state` are terminal on success.
Runtime input: `current_request`, `active_state`, and `history_index`.
Output: Return exactly one structured tool call. Inspection is non-terminal; commit and reuse are terminal. Fields marked optional may be omitted, and no additional fields are allowed.

```
inspect_state_forest
{"tree_id":"<string, optional>",
"state_id":"<string, optional>"}
commit_resolved_query
{"base_state_id":"<string or null>",
"resolved_query":"<complete requirement>",
"summary":"<short change summary>"}
reuse_resolved_state
{"state_id":"<string>",
"summary":"<short reuse summary>"}
```

I.2 Downstream Execution Routing

When a verified workspace is available, the following prompt decides whether the downstream executor should solve from scratch, reuse and patch the workspace, or inspect it before deciding. The `change_hint` is a non-authoritative summary generated by the Git Agent from model-visible interaction history; it contains no evaluator reference, gold answer, or gold edit.

Downstream Execution Router Prompt

You choose how the downstream executor should complete a new resolved task when a verified base workspace is available. Use exactly one routing tool per response.
The `current_resolved_task` is authoritative. The `base_resolved_task` describes what the existing workspace was built for. `change_hint` is only a hint, never a deterministic patch. Prefer correctness over reuse. Consider recomputation cost, validation cost, stale-work risk, and how much verified work remains applicable.

- `select_fresh_solve`: independently solve the complete current task. Prefer this when recomputation is cheap, inherited reasoning is risky, or checking old work costs as much as rebuilding it. Short calculations are often safer to recompute.
- `select_reuse_and_patch`: fork the verified base workspace and update only what the complete current task requires. Prefer this when valuable reports, research, evidence, code structure, style, or interfaces would be expensive to recreate. Set `preserve_base_calculations=true` only when verified base calculations are supplied and every supplied calculation remains required and unchanged. It may be true when the task adds new calculations, but must be false if any base calculation is replaced, withdrawn, stale, or no longer required.
- `inspect_base_workspace`: read one relevant file before deciding. Inspect only when the task comparison and manifest are insufficient. After inspection, select a final strategy. If uncertainty remains, call `select_fresh_solve`.

These are considerations, not fixed domain rules. Do not choose reuse merely because a base exists. Do not output Runtime IDs, patches, verification claims, or an answer to the underlying task. A final strategy tool call ends routing.
Runtime input: `current_resolved_task`, `base_resolved_task`, `change_hint`, and the base-workspace manifest.
Output: Return exactly one structured tool call per response. Inspection is non-terminal and must be followed by a selection call; either selection call is terminal. Fields marked optional may be omitted, and no additional fields are allowed.

```
inspect_base_workspace
{"path":"<string>", "offset":<integer ≥ 0, optional>,
"limit":<integer in [1,12000], optional>,
"reason":"<string>"}
select_fresh_solve
{"reason":"<non-empty string>"}
select_reuse_and_patch
{"reason":"<non-empty string>",
"preserve_base_calculations":<true or false>}
```

I.3 Workspace Execution

The following shared suffix governs how the downstream executor reads and updates the selected workspace. Domain-specific prefixes define the available tools, artifact schema, and completion conditions and are provided separately.

Shared Workspace Method-Selection Suffix

Treat `current_resolved_task` as the complete current task specification. Choose the working method yourself before acting. Do not read prior workspace content merely because it exists. First decide whether the task is self-contained and cheap to complete inde-

pendently. For a small math problem, short question, or small generation task, solve directly from `current_resolved_task` without inspecting prior output. Read prior files only when the task depends on valuable accumulated work such as existing code structure, report content, references, style, or interfaces that would be wasteful or unsafe to recreate. For a localized change to such work, inspect only relevant files and patch the smallest necessary region. For a structural change, use existing files as reference and rewrite only affected artifacts. If existing work already satisfies the task, finish without editing. Do not report this choice as Runtime metadata; express it through your tool actions.
Runtime input: `current_resolved_task`, the selected workspace, its manifest, and any domain-specific execution context.
Output: Return exactly one available structured tool call per response. The shared workspace calls are shown below; a domain may restrict paths or add a task tool such as `calculate` or `browsecomp_search`. No additional fields are allowed.

```
glob_workspace
{"pattern":"<string>"}
grep_workspace
{"pattern":"<string>",
"paths":<array of strings, optional>}
read_workspace
{"path":"<string>", "offset":<integer ≥ 0, optional>,
"limit":<integer in [1,50000], optional>}
write_workspace
{"path":"<string>", "content":"<string>"}
patch_workspace
{"path":"<string>",
"replacements":[{"old_text":"<string>",
"new_text":"<string>"}]}
delete_workspace
{"path":"<string>"}
finish_work_memory
{"final_response":"<string>"}
```

I.4 Evaluation Prompt

The following frozen prompt is used only to evaluate whether the final resolved query expresses the same complete task as the hidden reference. It is not shown to the Git Agent and does not generate the downstream task answer. At runtime, {`domain_specific_semantic_check`} is filled from Table 12.

Query-Equivalence Judge Prompt

Judge whether `PREDICTION` and `REFERENCE` express the same complete current task. Account for the requested goal, all active facts, constraints, comparisons, output requirements, withdrawals, and replacements.
Judge semantic effect rather than textual minimality. Equivalent paraphrases are correct. Extra accepted background is allowed only when it is non-conflicting and cannot change the requested goal, answer, scope, or active constraints. Do not apply this allowance to stale or withdrawn clauses, or to facts that could affect the answer or scope.
Missing active clauses, retaining material stale or withdrawn clauses, changing values, or introducing contradictions are not equivalent. Ignore differences in purely decorative step-by-step or boxed-answer wrappers; enforce explicit structured-output fields, precision, and task constraints.
Domain-specific semantic check: {`domain_specific_semantic_check`}
Input (JSON):

```
{
"prediction":  "{candidate_resolved_query}",
"reference":  "{hidden_reference}"
}
```

Output: Return exactly one `judge_query_equivalence` tool call.

```
{
"equivalent":  <true or false>,
"error_type":  "<none | missing | stale | contradictory |
different_goal>"
}
```

Do not solve the task.