LOOPSBENCH: FROM HARNESS ENGINEERING TO LOOP ENGINEERING IN CODING AGENT EVALUATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Long-horizon coding requires agents to implement interdependent requirements, track unfinished work, and preserve completed functionality. Existing benchmarks largely emphasize final outcomes, providing limited insight into this process. We introduce LOOPSBENCH, a benchmark for evaluating loop engineering: the mechanisms that organize agent work across successive steps. It comprises 112 tasks across 8 programming languages and 9 domains, with more than 5,300 separately testable development units organized into dependency DAGs recovered from source materials. The evaluator activates each unit's hidden tests once its prerequisite units pass their tests and reruns activated tests on subsequent code snapshots to track progress and regressions. We also analyze recorded plans, code changes, test activity, and evaluation outcomes. Across the evaluated models and harnesses, external continuation generally improves resolution. The strongest configuration, Opus-4.7 with Claude Code and external continuation, resolves 25.00% of tasks. Further analysis shows incomplete agreement between recorded plans and recovered dependencies, longer patches than the gold reference on resolved units, and regressions across the four configurations examined in the execution analysis.

1 INTRODUCTION

Coding agents increasingly undertake software development that requires implementing interdependent requirements, tracking unfinished work, and preserving existing functionality (Carlini, 2026). We refer to sustained development across such requirements as *long-horizon coding*. On top of the agent harness, mechanisms such as Codex goal mode and Claude Code's goal mode and dynamic workflows help organize work toward a continuing objective (OpenAI, 2026a; Anthropic, 2026a;c;b). We use *loop engineering* to describe the design of these mechanisms: how an agent chooses what to do next, tracks progress, and carries relevant information across successive steps.

Limitations of Existing Benchmarks. Existing benchmarks capture this setting only in part. Extensions of SWE-bench (Jimenez et al., 2024) cover additional languages, multimodal inputs, and larger feature changes (Zan et al., 2025; Yang et al., 2024b; Deng et al., 2025). Feature-level benchmarks (Zhou et al., 2026; Chen et al., 2025; Thai et al., 2025) further increase the scope of requested changes. However, evaluation still largely focuses on whether the final output resolves an issue or implements a requested feature. Final outcomes alone do not show when individual requirements were completed, whether later edits broke previously working functionality, or whether work began before its prerequisites were satisfied. Evaluating this process requires tracking requirements and their dependencies throughout execution.

LOOPSBENCH as a Benchmark for Loop Engineering Evaluation. LOOPSBENCH (Figure 1) represents each task as a set of separately testable parts, which we call *development units*. We recover these units and their prerequisite relationships from source materials and organize them into a directed acyclic graph (DAG): each node represents a unit, and each edge records a source-evidenced prerequisite relation. We use the recovered graph to analyze work order without assuming that it specifies an optimal order.

During execution, the evaluator activates a unit's hidden tests once all its prerequisite units pass their tests. Activated tests remain active on subsequent code snapshots, allowing progress and regressions to be tracked over time. The agent's plans, code changes, test activity, and evaluation results form a

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.

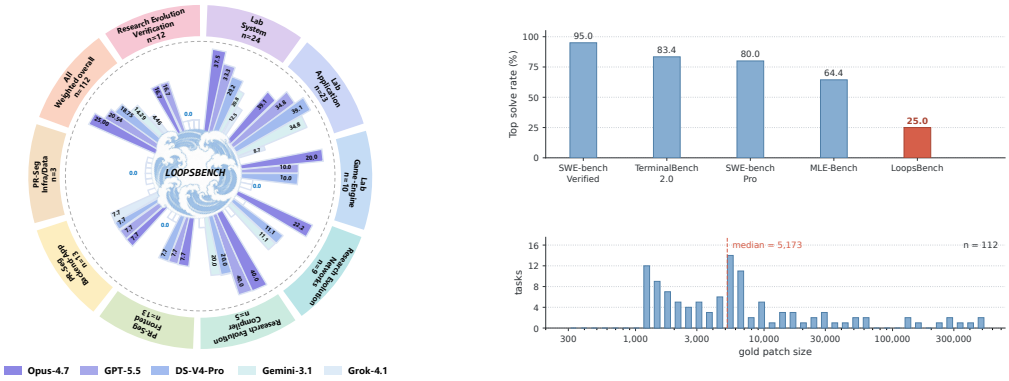

Figure 1: Summary statistics and results for LOOPSBENCH.

Table 1: Comparison of LOOPSBENCH with existing coding benchmarks. **Verifier**: basis for verification (F2P: fail-to-pass tests); **Lng**: number of programming languages; **Task Form**: task representation; **Units**: whether intermediate units are separately testable (partial denotes limited exposure); **DAG**: whether prerequisites between units are explicit. **Pch** and **Tst** report average reference-patch size (added plus deleted lines) and test count per task. **Time** reports the stated task or source time span, in days (d), months (m), or years (y). For LOOPSBENCH, it is the source-artifact calendar span defined in Section 2.3.

Benchmark	Verifier	Lng	Task Form	Units	DAG	Pch	Tst	Time
LOOPSBENCH	loop trace metrics	8	dependency DAG	✓	✓	37,296	74	6.6m
SWE family								
SWE-bench	end state F2P	1	issue	✗	✗	33	9	24.6d
Multi-SWE-bench	end state F2P	7	issue	✗	✗	107	21	18.5d
SWE-bench Pro	end state F2P	11	issue or feature	✗	✗	462	38	1.7m
FeatureBench	end state F2P	1	feature	*partial*	✗	1,256	38	19.5d
SWE-EVO	end state F2P	1	software evolution	*partial*	✗	610	31	7.1d
Long horizon family								
MLE-Bench	competition score threshold	1	repo generation	✗	✗	514	1	3.0m
NL2Repo-Bench	end state F2P	1	repo generation	*partial*	✗	1,623	247	3.4y
ProgramBench	end state F2P	1	repo generation	*partial*	✗	2,104	52	12.4d
RepoZero	end state F2P	1	repo generation	*partial*	✗	3,287	68	21.7d

loop trace. Test activation affects scoring; agents choose their own work order and do not receive checkpoint results.

LOOPSBENCH contains 112 tasks spanning 9 domains and 8 programming languages: 29 tasks built from sequences of merged pull requests (PR Sequences), 57 from university programming projects (Course Labs), and 26 from research methods that build on earlier work (Research Evolutions). The PR Sequence tasks were selected from an initial pool of 56 GitHub repositories. Together, these tasks contain more than 5,300 development units. The median dependency depth is 4, where depth measures the length of the longest prerequisite chain within a task. Table 1 compares whether benchmarks provide separately testable parts within each task and explicitly represent their dependencies. To our knowledge, LOOPSBENCH is the first benchmark for loop engineering evaluation to combine explicit dependency graphs, dependency-aware testing, and execution traces.

Takeaways. Three findings emerge. First, Resolve Rate varies across model and harness configurations and generally increases with external continuation; the strongest configuration resolves 25.00% of tasks. Second, recorded plans only partly match the recovered dependencies, patches for successfully completed units are longer than the gold reference, and the task-averaged fraction of

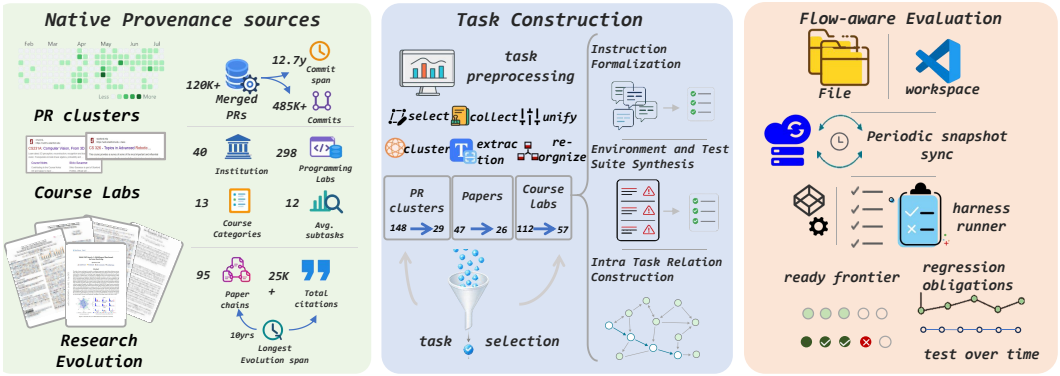

Figure 2: Overview of the LOOPSBENCH construction and evaluation pipeline. Development units and the dependency DAG are recovered before test synthesis.

authored tests distinguishing the initial state from the gold solution is below one half. Third, context renewal differs across the four configurations examined in the execution analysis, and previously passing tests fail again in each configuration.

Contributions. Our contributions are threefold:

- **(i)** We construct LOOPSBENCH, a benchmark of 112 long-horizon coding tasks across 8 programming languages and 9 domains, through a pipeline that recovers testable development units and their dependencies from source artifacts and packages reproducible execution environments.
- **(ii)** We introduce a dependency-aware evaluation protocol that activates hidden tests according to prerequisite state and retains them for regression checks on later code snapshots, producing traces of development throughout execution.
- **(iii)** We evaluate frontier models and widely used agent harnesses across loop configurations, analyzing dependency planning, implementation, test authoring, and context renewal alongside task resolution.

2 LOOPSBENCH AS A PIPELINE FOR LOOP ENGINEERING EVALUATION IN LONG HORIZON CODING

Figure 2 summarizes the five stages used to construct LOOPSBENCH tasks. **(1) Task Collection** gathers materials from university programming projects, GitHub repositories, and research papers. **(2) Task Preprocessing** organizes these materials into candidate tasks. **(3) Task Selection** selects candidates based on the calendar span recorded in the source materials and the size of the reference code changes. **(4) Development Units and Dependency Recovery** identifies separately testable parts within each task and uses evidence from the source materials to determine which parts depend on others. **(5) Task Instrumentation** prepares the instructions given to the agent, a reproducible execution environment, and tests for each development unit. The resulting tasks let us track which requirements an agent completes and whether later changes break previously working functionality.

2.1 TASK COLLECTION

We collect materials from three types of sources covering different programming domains, task structures, and dependency depths. All resulting tasks use the same evaluation procedure. Appendix A.5 lists the released tasks.

(1) Course Labs. We collect 112 programming projects from 30 universities through official course websites and public GitHub copies of released assignments. For each project, we collect the instructions, starter code, and instructor-provided reference tests.

(2) PR Sequences. We collect the full commit histories and merged pull requests of 56 actively maintained GitHub repositories. These repositories have many GitHub stars and a long history of community maintenance.

(3) Research Evolutions. We collect 212 highly cited papers from 48 publication venues, each with a fully open-source official implementation under a license approved by the Open Source Initiative (OSI). We also collect the LaTeX source files available on arXiv for the earlier work these papers cite and for later papers that cite them. We then retain 95 papers connected by methodological dependencies, meaning that one paper builds directly on another's method.

2.2 Task Preprocessing

We organize the collected materials into candidate tasks before applying the selection criteria in the next subsection. The steps differ by source, but all preserve the evidence needed to identify dependencies within each task. Appendix A.4 gives a concrete materialized task example.

Course Labs. We translate and combine project instructions written in different languages into a single English version. We then organize the starter code under one task directory. Each project becomes one candidate task, yielding 112 candidates.

PR Sequences. We clone each repository with its full commit history. For each merged pull request, we extract the code changes and separate them into test code, implementation code, and supporting files. We then use Claude Code (Anthropic, 2025d) to inspect the repository and group changes into candidate tasks, which are checked against a fixed set of acceptance criteria. Each candidate uses a task-entry snapshot as its starting codebase. This process yields 148 candidates from the 56 repositories.

Research Evolutions. We start with an influential early paper from the collected publication venues. We retrieve its LaTeX source from arXiv, along with the sources of papers it cites and papers that cite it. We use Claude Code to distinguish general related-work citations from cases where one paper builds directly on another's method. We add papers linked by these methodological dependencies to the group, then examine the citations of each newly added paper. We repeat this process until no additional papers qualify. Each group becomes one candidate task, yielding 47 candidates before selection.

2.3 Task Selection

To focus on substantial development tasks, we retain candidates that meet both of the following thresholds. After filtering, 112 tasks remain in the released benchmark: 57 Course Labs, 29 PR Sequences, and 26 Research Evolutions.

Temporal Span. The time span recorded in the source materials must be at least 2.5 months. For Course Labs, we use the project duration specified in the instructions. For PR Sequences, we use the time between the merges of the first and last commits in the candidate's commit sequence. For Research Evolutions, we use the interval between the publication dates of the earliest and latest papers in the group. **Solution Scale.** The reference patch must contain at least 1,200 changed lines, counting both additions and deletions. Appendix A.5 reports per-task structural and implementation statistics.

2.4 Development Units and Dependency Recovery

A development unit is a distinct part of a task with its own requirement and tests. We identify these parts from the source materials. For PR Sequences, units are merged pull requests or smaller portions supported by source evidence. For Course Labs, they are released modules or milestones. For Research Evolutions, they are additions to earlier research methods. We represent each unit as $u = (r_u, s_u, p_u, \Delta_u, T_u)$, where r_u is its requirement, s_u identifies the files or code symbols it concerns, p_u lists its prerequisite units, Δ_u contains its reference code changes, and T_u is its set of standard tests.

We formalize each task as a DAG $G = (V, E)$, where V is the set of development units and E is the set of dependency edges. The graph lets us observe how a loop orders work in relation to unit-level prerequisites. We use the recovered DAG as an evaluation contract rather than a complete causal model of software development. The four admitted patterns are *sequential PR chains* along the merged commit history, *structural module reuse* when v edits a file or symbol that u introduced, *functional producer consumer API edges* when v calls or imports an authoritative definition created by u, and *compositional layering* when v extends a subclass, schema, or interface declared by u.

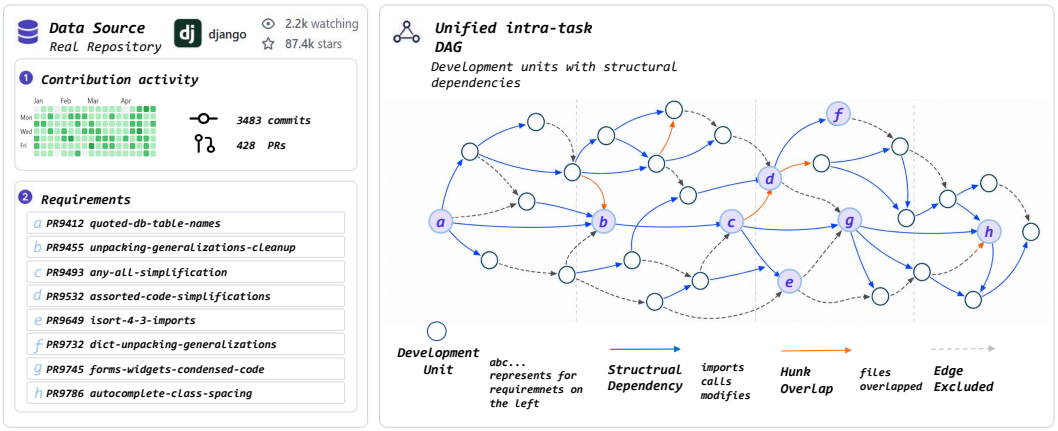

Figure 3: Four prerequisite patterns in a materialized task.

These patterns describe dependency relationships, while the reference code and patches provide evidence for adding edges. We add an edge from unit u to unit v only when this evidence shows that v depends on u. We examine overlapping changes to the same region of a file and symbol-level relationships in which v imports or calls a symbol introduced by u, or inherits from a class introduced by u. Historical order alone is not sufficient to establish an edge. Before adding edges, we exclude generated files, dependency lockfiles, and files modified by most pull requests in the repository. Changes to these files do not create dependency edges, since frequent or automatically generated edits can otherwise create misleading dependency chains.

We retain only semantically separable, independently testable units. Their tests must distinguish their own functionality from that of their prerequisites and pass after the unit's reference changes are added, without any successor changes. Section 2.5 describes the checks used to validate these conditions. If changes cannot be separated into a distinct, testable unit, we assign them to the nearest prerequisite unit or merge them with another unit. The reference patch is used for benchmark construction and analysis, but it is not treated as the only valid implementation or way to divide the task.

Agents can revise earlier code, but completed requirements must remain satisfied. Tasks in which later development replaces or removes earlier requirements are outside the scope of this benchmark. Section 2.6 describes how the DAG is used during evaluation.

2.5 Task Instrumentation

For each selected task, we prepare instructions, a reproducible execution environment, and tests of the required functionality. This construction is performed offline with assistance from Claude Code. We use the complete reference implementation, or gold solution, to construct and validate the task and compute reference statistics before evaluation.

❶ **Instruction Formalization.** PR titles, course project instructions, and paper method sections often leave some expected behavior unspecified. Claude Code (Anthropic, 2025d), backed by Claude Opus 4.7 (Anthropic, 2026d), uses the differences between the starting codebase and the reference implementation to identify the requirements and produce a single task instruction. The instruction specifies what to implement without revealing the reference implementation's exact edit locations or patch structure. It retains public function signatures and expected output formats needed to satisfy the tests, including code symbol names required by test assertions. We remove repository names, course identifiers, paper titles, and author names to reduce recognition of the original source from the prompt.

❷ **Environment and Test Suite Synthesis.** Claude Code identifies the software dependencies in the task materials and writes a `Dockerfile`, together with a `docker-compose.yaml` file for tasks involving multiple services. We process units in topological order, so tests for later units are written against the prerequisite state used for test activation. Across all sources, the evaluator runs the standard tests through a common entry point, `run-tests.sh`, which wraps the task's test runner. This evaluator entry point is separate from the agent's use of the project's existing tests.

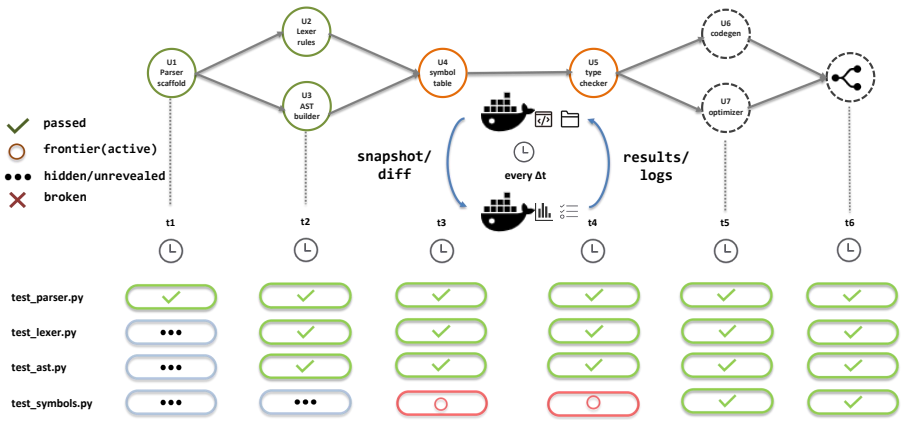

Figure 4: Dependency-aware evaluation protocol.

For each unit u, we draft missing tests and apply its ancestors' reference patches to the starting codebase in topological order, rerunning T_u after each patch. At least one test must still fail after all ancestor patches have been applied and before u's patch is added. All tests must pass after adding u's reference patch, with successor patches unapplied. These checks distinguish the unit's own functionality from that of its prerequisites and establish that it can be tested without later work. In addition, every test must fail on the starting codebase (*non-triviality*) and pass on the complete reference implementation (*solvability*).

If a test produces no execution result, we treat this as an environment issue and revise the `Dockerfile`. If it runs but fails on the reference implementation, we revise its assertions. After processing all units, we apply the full reference patch to the starting codebase and run the combined test suite. Units with unresolved environment or test issues are discarded when the construction budget is exhausted. These executable checks are followed by human review of semantic agreement between the source requirements and the tests (Appendix A.2); Appendix G discusses the remaining limits of validation and construction bias.

2.6 Evaluation Protocol

As illustrated in Figure 4, the agent receives the overall task instruction and files containing each unit's requirement, relevant files or code symbols, and the conditions each unit must satisfy. It can inspect the starting codebase, run existing project tests, write its own tests, and view command outputs. The evaluator withholds prerequisite lists, the full DAG, standard tests, the reference implementation and patches, construction records, the ready frontier, checkpoint results, and scoring state. Table 5 in Appendix A.1 summarizes this separation. We record the order in which the agent works on units and compare its planned dependencies with the recovered DAG. Any ordering consistent with the DAG is valid, not just the original PR order.

Dependency-Aware Test Activation. The dependency graph determines when the evaluator begins running each unit's standard tests, which remain hidden from the agent. At each checkpoint, a unit is marked complete if all of its standard tests pass. The *ready frontier* contains units that are not currently complete but whose prerequisite units are all complete. For the task graph G, the ready frontier at checkpoint t is

$$R_t \;=\; \{\, v \in V \setminus C_t \;:\; \forall (u,v) \in E,\; u \in C_t \,\},$$

where C_t is the set of units whose tests all pass at t.

The evaluator activates ready-frontier tests to avoid unnecessary downstream evaluation and mirror the progressive introduction of tests as prerequisite functionality becomes available in development. Within each frozen snapshot, it repeatedly activates and tests all newly ready units until no further unit can be activated; final evaluation follows the same procedure. Activated tests are rerun at every subsequent checkpoint, including after unit or prerequisite failures, to track progress and regressions. Checkpoint results remain hidden from the agent. These rules control evaluation, not what the agent is allowed to edit. The agent chooses its own work order without being told which units are ready for evaluation. Test activation controls which obligations are evaluated, while Test Pass Rate always uses

the complete task suite as its fixed denominator. Unreleased tests receive no passing credit, and task resolution requires every test in the complete suite to pass (Appendix C.1).

Two-Container Evaluation. We use separate containers for the agent's work and for evaluation, so the agent can continue editing while hidden tests run. Container A holds the working codebase and is the only container the agent can modify. Container B holds the full standard test suite and the reference execution environment. It evaluates snapshots of the code from Container A, where each snapshot is a copy of the working codebase at a particular time.

At fixed intervals, the evaluator samples Container A and checks whether accumulated code changes meet the configured size threshold (Appendix B.4). Qualifying snapshots are queued in creation order and sent to Container B, which runs all standard tests activated so far against each frozen snapshot. The results track progress and regressions over successive code states as well as the final state.

3 EXPERIMENTS

We organize the experiments around three questions about coding agent loops. All experiments use the full 112 task release of LOOPSBENCH.

3.1 RQ1: MODELS, HARNESSES, AND CONTINUATION

What performance differences does LOOPSBENCH *reveal across models, harnesses, and continuation settings?*

Table 2 compares models within each family using their vendor's harness, different models under Claude Code, and different harnesses with a fixed `gpt-5.4` model. Each comparison includes execution with and without external continuation, which starts fresh agent invocations while preserving the workspace. Resolve Rate (RR) measures full task completion, and Test Pass Rate (TPR) measures the fraction of each task's complete, fixed test suite that passes. Depth measures progress at the end of the first execution round: the deepest dependency layer whose tests and all preceding layers' tests pass, normalized by task depth and averaged across tasks. Appendix C.2 reports supplementary within-family pass@3 results over three task attempts; Appendix C.3 reports mean billed tokens per task.

The results reveal performance differences across model families and harnesses that remain visible under continuation. Within each family, the model ordering agrees across RR, TPR, and Depth; model differences persist under a fixed harness, and harness differences persist under a fixed model. Codex leads the fixed-model harness comparison, while Opus-4.7 with Claude Code achieves the strongest overall completion and dependency progress. Continuation improves TPR for every configuration and RR for all except Qwen2.5-72B, which resolves no tasks in either setting. Its higher TPR nevertheless exposes progress that task resolution alone misses. Together with Depth, these measurements let LOOPSBENCH distinguish partial development progress even when complete solutions remain out of reach.

> **RQ1:** LOOPSBENCH distinguishes performance across model families, harnesses, and continuation settings through both task completion and partial dependency progress, enabling comparisons even when tasks remain unresolved.

3.2 RQ2: DEVELOPMENT BEHAVIORS

What development behaviors does LOOPSBENCH *reveal beyond task completion?*

Table 3 compares development behavior across six harnesses through recorded plans, code changes, authored tests, and checkpoint outcomes. Agents are instructed to record a plan DAG before editing and update it when they revise dependencies (Appendix B.3). We compare available final plan DAGs with the recovered DAG, examine patches on units resolved by each configuration, and track test authoring and regressions throughout execution. The planning comparison allows any ordering consistent with the recovered DAG.

The results reveal differences in how agents plan, implement, and test their work. **Planning.** Edge F1 and Layer ρ measure agreement with the recovered edges and topological layers; both show incomplete agreement with the reference graph. Critical Path Ratio (CPR), the fraction of nodes on a plan's longest path, and Width Ratio (WR), its maximum topological layer width relative

Table 2: RQ1 performance across models and agent harnesses. "w/o outer" and "w/ outer" denote execution without and with outer continuation. Per task billed tokens are reported in Appendix C.3.

Model	RR↑		TPR↑		Depth↑
	w/o outer	*w/ outer*	*w/o outer*	*w/ outer*	
GPT (Codex)					
GPT-5.5	14.29%	21.43%	39.62%	50.84%	0.53
GPT-5.2	9.82%	13.39%	33.91%	43.90%	0.36
GPT-5	5.36%	8.04%	28.45%	36.86%	0.22
GPT-4o	1.79%	3.57%	21.85%	28.30%	0.09
Claude (Claude Code)					
Opus-4.7	16.96%	25.00%	41.18%	53.05%	0.61
Opus-4.5	12.50%	17.86%	36.97%	48.21%	0.46
Sonnet-4.5	8.93%	12.50%	32.85%	42.58%	0.34
Sonnet-4	4.46%	7.14%	27.43%	35.54%	0.18
Qwen (Qwen Code)					
Qwen3.6-Plus	6.25%	9.82%	37.97%	45.10%	0.31
Qwen3.5-Plus	5.36%	8.04%	31.75%	41.17%	0.19
Qwen3-Max	0.89%	1.79%	17.45%	23.31%	0.07
Qwen2.5-72B	0.00%	0.00%	10.61%	13.88%	0.03

	RR↑		TPR↑		Depth↑
	w/o outer	*w/ outer*	*w/o outer*	*w/ outer*	
Model *(fixed harness: Claude Code)*					
Opus-4.7	16.96%	25.00%	41.18%	53.05%	0.61
GPT-5.5	13.39%	20.54%	38.04%	49.30%	0.47
GLM-5.1	13.39%	18.75%	34.55%	45.49%	0.44
DeepSeek-V4P	11.61%	18.75%	36.07%	47.34%	0.46
Gemini-3.1-Pro	8.93%	14.29%	32.86%	43.59%	0.31
Qwen3.6-Plus	6.25%	9.82%	36.43%	47.50%	0.34
Kimi-2.6	3.57%	6.25%	21.27%	28.33%	0.14
Grok-4.1-FR	2.68%	4.46%	23.32%	30.97%	0.15
Harness *(fixed model:* `gpt-5.4`*)*					
Codex	13.39%	18.75%	37.84%	49.06%	0.49
Claude Code	12.50%	17.86%	36.97%	48.21%	0.42
GitHub Copilot	10.71%	15.18%	34.21%	45.46%	0.38
OpenHands	6.25%	9.82%	29.21%	39.30%	0.20
SWE-agent	5.36%	8.93%	28.07%	38.11%	0.16
mini-swe-agent	4.46%	7.14%	25.96%	35.20%	0.13

to the reference, further distinguish plan structure. OpenHands, SWE-agent, and mini-swe-agent produce nearly linear, narrow plans, whereas Claude Code, Codex, and GitHub Copilot have shorter relative critical paths and widths closer to the reference. Thus, similar overall width can coexist with incomplete dependency agreement; these graph statistics do not measure actual execution concurrency.

Implementation. PatchLen measures patch length relative to the reference, and Jacc measures token-level Jaccard distance. Across configurations, patches on resolved units are longer than their references and differ substantially in token content. Successful unit completion therefore coexists with distinct implementation choices that a pass/fail result does not characterize. Each configuration resolves its own subset of units, and the reference patch is one valid implementation rather than a code-quality standard.

Table 3: Planning, implementation, and testing metrics with GPT-5.4 and outer continuation. Planning means use valid recorded DAGs; implementation metrics cover resolved units; Jacc is a distance.

	Planning fidelity				Implementation		Testing		
Harness	**Edge F1**	**Layer ρ**	**CPR**	**WR**	**PatchLen**	**Jacc**	**#T**	**F2P**	**Reg↓**
Claude Code	0.71	0.65	0.31	1.08	1.58	0.62	28	0.47	7.11%
Codex	0.67	0.61	0.33	1.14	1.71	0.64	24	0.44	4.83%
GitHub Copilot	0.58	0.52	0.41	0.92	1.83	0.66	22	0.41	6.91%
OpenHands	0.39	0.33	0.85	0.39	2.31	0.73	16	0.34	2.46%
SWE-agent	0.37	0.31	0.88	0.36	2.36	0.74	15	0.33	2.18%
mini-swe-agent	0.27	0.22	0.97	0.24	2.54	0.76	11	0.28	0.24%

Testing. #T is the mean number of authored tests per task; F2P is the task-averaged fraction failing on the initial workspace and passing on the gold solution. F2P remains below one half for every configuration. All six also show regressions: previously passing standard tests fail at later checkpoints. Regression Rate (Reg) pools these pass-to-fail transitions across all tasks and divides by total obligation exposure: previously passing tests counted at each later checkpoint. It must be read alongside progress and exposure. Appendices D.2 and C.1.4 provide metric details.

> **RQ2:** LOOPSBENCH makes dependency planning, implementation choices, and test authoring observable alongside the preservation of working functionality, exposing development behaviors that final completion scores alone cannot describe.

3.3 RQ3: EXECUTION PROFILES

What execution profiles does LOOPSBENCH *reveal across loop mechanisms?*

Table 4 compares Codex goal mode (OpenAI, 2026a), Claude Code goal mode (Anthropic, 2026a), Claude dynamic workflows (Anthropic, 2026c;b), and Ralph's outer continuation (Huntley, 2025; 2026). Codex and Ralph use GPT-5.5; the Claude configurations use Opus-4.7. Ralph invokes mini-SWE-agent for outer continuation. Rounds counts outer invocations plus context segments opened by compaction, extracted from trajectories. Reg/run counts snapshots with newly regressed tests, averaged over all 112 runs. Dynamic workflow Rounds covers only the coordinator and its restarts;

regression checks include worker edits. Shorter and Longer analytically rescale the compaction interval at fixed context use. Appendices E.1.1–E.1.4 detail definitions, calculation, and coverage.

All four profiles show regressions. Dynamic workflows have the highest observed RR (27 resolved tasks) and the most coordinator rounds. Regressions occur at both high and low round counts. At fixed context use, shorter compaction intervals yield more rounds; this calculation does not predict regression or resolution changes. The comparison spans different models and loop mechanisms and does not establish that context renewal causes regressions.

Table 4: Execution profiles on 112 tasks. Shorter and Longer denote compaction intervals 25% below and above the default, respectively, at fixed context use. Dynamic workflow rounds cover only the coordinator. Reg/run counts snapshots with newly regressed tests per run.

Loop mechanism	Default execution			Rounds by interval	
	Rounds	Reg/run	RR (%)	Shorter	Longer
Codex goal mode	32.76	0.34	20.54	43.6	26.3
Claude Code goal mode	34.69	0.13	17.86	46.2	27.8
Claude dynamic workflows	97.96	0.38	24.11	130.6	78.4
Ralph loop	13.24	0.17	8.04	17.2	10.8

RQ3: LOOPSBENCH characterizes context renewal and tracks the preservation of previously passing tests, making loop execution itself an evaluation target alongside final task success.

4 RELATED WORK

Software Engineering Benchmarks. SWE-bench (Jimenez et al., 2024) targets repository repair; FeatureBench and SWE-EVO (Zhou et al., 2026; Thai et al., 2025) cover feature development and evolution with partial-credit and preservation checks. ProgramBench (Yang et al., 2026) reconstructs programs from executable access and documentation; RepoZero (Zhang et al., 2026) evaluates repository reproduction from API specifications and examples. LOOPSBENCH evaluates progress through development-unit prerequisites and tracks whether later edits preserve newly completed functionality, distinguishing execution outcomes even on unresolved tasks.

Coding Agents and Execution Loops. SWE-agent and OpenHands (Yang et al., 2024a; Wang et al., 2025) exemplify open-source harnesses; industrial agents include Claude Code, Codex, GitHub Copilot, Cursor, and Devin (Anthropic, 2025d; OpenAI, 2025a; GitHub, 2026; Anysphere, 2026; Cognition, 2026). Goal modes, dynamic workflows, and Ralph loops (OpenAI, 2026a; Anthropic, 2026a;c; Huntley, 2025) support persistent objectives, worker orchestration, and repeated invocations. Under a shared task and scoring contract, LOOPSBENCH compares dependency planning, functionality retention, and context renewal alongside task success across harnesses and continuation policies. Appendix F expands the comparison.

5 CONCLUSION

LOOPSBENCH makes loop engineering a measurable dimension of coding agent evaluation alongside end state scoring. It formalizes long horizon tasks as dependency DAGs, activates tests when all prerequisite units pass their tests, and reruns activated tests to detect regressions while leaving execution order to the evaluated loop. The resulting traces show how agents handle unfinished work, maintain state, and preserve working functionality across iterations. Across frontier LLMs and loop implementations, the strongest configuration resolves 25.00% of tasks, regression events remain visible across all four RQ3 loop profiles, and the recorded plans recover only part of the prerequisite DAG. This comparison allows any valid topological order and does not require agents to follow the historical PR sequence. These findings motivate reporting task outcomes alongside trace measurements of planning, residual work, and regression across model and loop configurations. Scope conditions, source assumptions, and version pinning caveats are stated in Appendix G.

COMPLIANCE, SAFETY, AND DATA STATEMENT

LOOPSBENCH is constructed from public open source repositories, public course artifacts, and publicly released research code. It does not use personal data, customer data, or private production data. Generated benchmark instances are derived from source artifacts and should preserve the applicable source licenses, attribution requirements, and redistribution constraints. The construction pipeline may reintroduce historical defects, including security relevant defects, as part of benchmark generation. These instances are intended for research evaluation rather than deployment. We assume source histories and release artifacts are benign and trusted. Malicious, poisoned, or intentionally deceptive repositories are outside the scope of this benchmark.

AI USE STATEMENT

We used Claude Code, backed by Claude Opus 4.7, to assist with offline benchmark construction, including source inspection, requirement formalization and translation, dependency recovery, environment preparation, and test generation. The construction and validation procedures are described in Section 2 and Appendix A.2. LLM-based tools also assisted with manuscript revision, language editing, literature checking, and LaTeX preparation. The authors are responsible for the accuracy and integrity of the manuscript, benchmark artifacts, and reported results.

REFERENCES

Alibaba Cloud. Alibaba cloud unveils strategic roadmaps for the next generation AI innovations. https://alizila.oss-us-west-1.aliyuncs.com/uploads/2025/09/Media-Release-Alibaba-Cloud-Unveils-Strategic-Roadmaps-for-the-Next-Generation-AI-Innovations-0924.pdf, 2025. Includes Qwen3-Max. Accessed 2026-05-16.

Alibaba Cloud. Qwen3.6-Plus model information. https://www.alibabacloud.com/help/en/model-studio/qwen3-6-plus, 2026. Accessed 2026-09-25.

Alibaba Group. Alibaba open-sources Qwen3.5, a natively multimodal model built for high-efficiency inference. https://www.alibabagroup.com/document-1960233590314762240, 2026. Identifies Qwen3.5-397B-A17B as Qwen3.5-Plus. Accessed 2026-05-16.

Anthropic. System card: Claude Opus 4 & Claude Sonnet 4. https://assets.anthropic.com/m/6c940a1b69ed6a1c/original/Claude-4-System-Card.pdf, 2025a. Covers Claude Opus 4 and Claude Sonnet 4. Accessed 2026-05-16.

Anthropic. Introducing Claude Opus 4.5. https://www.anthropic.com/news/claude-opus-4-5, 2025b. Accessed 2026-05-16.

Anthropic. Introducing Claude Sonnet 4.5. https://www.anthropic.com/news/claude-sonnet-4-5, 2025c. Accessed 2026-05-16.

Anthropic. Claude code, 2025d. URL https://www.anthropic.com/claude-code. Anthropic product page.

Anthropic. Keep claude working toward a goal – claude code docs, 2026a. URL https://code.claude.com/docs/en/goal. Accessed: 2026-06-14.

Anthropic. Create custom subagents – claude code docs, 2026b. URL https://code.claude.com/docs/en/sub-agents. Accessed: 2026-06-14.

Anthropic. Orchestrate subagents at scale with dynamic workflows – claude code docs, 2026c. URL https://code.claude.com/docs/en/workflows. Accessed: 2026-06-14.

Anthropic. Introducing Claude Opus 4.7. https://www.anthropic.com/news/claude-opus-4-7, 2026d. Accessed 2026-05-16.

Anysphere. Cursor Agent, 2026. URL https://cursor.com/docs/agent/overview. Cursor Docs. Accessed: 2026-09-25.

Nicholas Carlini. Building a C compiler with a team of parallel claudes, 2026. URL https://www.anthropic.com/engineering/building-c-compiler. Anthropic Engineering blog.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Lilian Weng, and Aleksander Mądry. MLE-bench: Evaluating machine learning agents on machine learning engineering. *arXiv preprint arXiv:2410.07095*, 2024. URL https://arxiv.org/abs/2410.07095.

Guoxin Chen, Fanzhe Meng, Jiale Zhao, Minghao Li, Daixuan Cheng, Huatong Song, Jie Chen, Yuzhi Lin, Hui Chen, Xin Zhao, et al. BeyondSWE: Can current code agent survive beyond single-repo bug fixing? *arXiv preprint arXiv:2603.03194*, 2026. URL https://arxiv.org/abs/2603.03194.

Haorui Chen, Chengze Li, and Jia Li. FeatBench: Evaluating coding agents on feature implementation for vibe coding. *arXiv preprint arXiv:2509.22237v1*, 2025. URL https://arxiv.org/abs/2509.22237v1.

Cognition. Introducing devin, 2026. URL https://docs.devin.ai/get-started/devin-intro. Devin Docs. Accessed: 2026-09-25.

DeepSeek-AI. DeepSeek-V4 preview release. https://api-docs.deepseek.com/news/news260424, 2026. Accessed 2026-09-25.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, et al. SWE-Bench Pro: Can AI agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025. URL https://arxiv.org/abs/2509.16941.

Jingzhe Ding, Shengda Long, Changxin Pu, Huan Zhou, Hongwan Gao, et al. NL2Repo-Bench: Towards long-horizon repository generation evaluation of coding agents. *arXiv preprint arXiv:2512.12730*, 2025. URL https://arxiv.org/abs/2512.12730.

Nicholas Edwards, Yukyung Lee, Yujun Audrey Mao, Yulu Qin, Sebastian Schuster, and Najoung Kim. RExBench: Can coding agents autonomously implement AI research extensions? In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 16380–16417, 2026. URL https://aclanthology.org/2026.acl-long.745/.

Yukang Feng, Jianwen Sun, Zelai Yang, Jiaxin Ai, Chuanhao Li, et al. LongCLI-Bench: A preliminary benchmark and study for long-horizon agentic programming in command-line interfaces. *arXiv preprint arXiv:2602.14337*, 2026. URL https://arxiv.org/abs/2602.14337.

Dhruv Gautam, Spandan Garg, Jinu Jang, Neel Sundaresan, and Roshanak Zilouchian. RefactorBench: Evaluating stateful reasoning in language agents through code. In *International Conference on Learning Representations*, volume 2025, pp. 43131–43162, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/6b44ee74539ea77d6a0d50d468724371-Paper-Conference.pdf.

GitHub. Github copilot cloud agent, 2026. URL https://docs.github.com/en/copilot/how-tos/use-copilot-agents/cloud-agent. GitHub Docs. Accessed: 2026-09-25.

Google DeepMind. Gemini 3.1 Pro model card. https://deepmind.google/models/model-cards/gemini-3-1-pro, 2026. Accessed 2026-05-16.

Geoffrey Huntley. Ralph wiggum as a "software engineer", 2025. URL https://ghuntley.com/ralph/. Accessed: 2026-05-07.

Geoffrey Huntley. Everything is a ralph loop, 2026. URL https://ghuntley.com/loop/. Accessed: 2026-06-14.

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. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/edac78c3e300629acfe6cbe9ca88fb84-Paper-Conference.pdf.

Thomas Kwa, Ben West, Joel Becker, Amy Deng, Katharyn Garcia, Max Hasin, Sami Jawhar, Megan Kinniment, Nate Rush, Sydney Von Arx, Ryan Bloom, Thomas Broadley, Haoxing Du, Brian Goodrich, Nikola Jurkovic, Luke Harold Miles, Seraphina Nix, Tao Lin, Chris Painter, Neev Parikh, David Rein, Lucas Jun Koba Sato, Hjalmar Wijk, Daniel M. Ziegler, Elizabeth Barnes, and Lawrence Chan. Measuring AI ability to complete long software tasks, 2026. URL https://arxiv.org/abs/2503.14499v4.

Steffen Lehnert. A taxonomy for software change impact analysis. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*, pp. 41–50, 2011. doi: 10.1145/2024445.2024454. URL https://doi.org/10.1145/2024445.2024454.

Bowen Li, Wenhan Wu, Ziwei Tang, Lin Shi, John Yang, Jinyang Li, Shunyu Yao, Chen Qian, Binyuan Hui, Qicheng Zhang, et al. DevBench: A comprehensive benchmark for software development. *arXiv preprint arXiv:2403.08604v1*, 2024. URL https://arxiv.org/abs/2403.08604v1.

Mike Merrill, Alexander Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *International Conference on Learning Representations*, volume 2026, pp. 40903–40986, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/444a3737adaee10d86ad2ef5f74468e6-Paper-Conference.pdf.

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

Moonshot AI. Kimi K2.6 model card. https://huggingface.co/moonshotai/Kimi-K2.6, 2026a. Accessed 2026-09-25.

Moonshot AI. Kimi K2.6: Advancing open-source coding, April 2026b. URL https://www.kimi.ai/blog/kimi-k2-6. Published April 20, 2026. Accessed: 2026-09-25.

OpenAI. Hello GPT-4o, May 2024a. URL https://openai.com/index/hello-gpt-4o/. Published May 13, 2024. Accessed: 2026-09-25.

OpenAI. GPT-4o system card. https://cdn.openai.com/gpt-4o-system-card.pdf, 2024b. Accessed 2026-05-16.

OpenAI. Introducing swe-bench verified, 2024c. URL https://openai.com/index/introducing-swe-bench-verified/. OpenAI blog.

OpenAI. Introducing codex, 2025a. URL https://openai.com/index/introducing-codex/. OpenAI product announcement.

OpenAI. Introducing GPT-5. https://openai.com/index/introducing-gpt-5/, 2025b. Accessed 2026-05-16.

OpenAI. Update to GPT-5 system card: GPT-5.2. https://cdn.openai.com/pdf/3a4153c8-c748-4b71-8e31-aecbde944f8d/oai_5_2_system-card.pdf, 2025c. Accessed 2026-05-16.

OpenAI. Using goals in Codex: Persistent objectives for long-running work, 2026a. URL https://developers.openai.com/cookbook/examples/codex/using_goals_in_codex. OpenAI Cookbook. Accessed: 2026-09-25.

OpenAI. Introducing GPT-5.3 Instant, GPT-5.4 Thinking, and GPT-5.4 Pro. https://academy.openai.com/en/home/resources/latest-model, 2026b. Accessed 2026-05-16.

OpenAI. Introducing GPT-5.5. https://openai.com/index/introducing-gpt-5-5/, 2026c. Accessed 2026-05-16.

OpenAI. Why we no longer evaluate SWE-bench Verified, 2026d. URL https://openai.com/index/why-we-no-longer-evaluate-swe-bench-verified/. OpenAI research note.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-Gym. *arXiv preprint arXiv:2412.21139*, 2024. URL https://arxiv.org/abs/2412.21139.

Qwen Team. Qwen2.5: A party of foundation models!, September 2024a. URL https://qwenlm.github.io/blog/qwen2.5/. Published September 19, 2024. Accessed: 2026-09-25.

Qwen Team. Qwen2.5 technical report. https://arxiv.org/abs/2412.15115, 2024b. Accessed 2026-05-16.

Günther Ruhe and Moshood Omolade Saliu. The art and science of software release planning. *IEEE software*, 22(6):47–53, 2005. doi: 10.1109/ms.2005.164. URL https://doi.org/10.1109/ms.2005.164.

Atharva Sehgal, James Hou, Akanksha Sarkar, Ishaan Mantripragada, Swarat Chaudhuri, Jennifer J. Sun, and Yisong Yue. FormulaCode: Evaluating agentic optimization on large codebases, 2026. URL https://arxiv.org/abs/2603.16011.

Zachary S Siegel, Sayash Kapoor, Nitya Nadgir, Benedikt Stroebl, and Arvind Narayanan. CORE-Bench: Fostering the credibility of published research through a computational reproducibility agent benchmark. *arXiv preprint arXiv:2409.11363*, 2024. URL https://arxiv.org/abs/2409.11363.

Atharv Sonwane, Eng-Shen Tu, Wei-Chung Lu, Claas Beger, Carter Larsen, Debjit Dhar, Simon Alford, Rachel Chen, Ronit Pattanayak, Tuan Anh Dang, et al. OmniCode: A benchmark for evaluating software engineering agents. *arXiv preprint arXiv:2602.02262*, 2026. URL https://arxiv.org/abs/2602.02262.

Minh V. T. Thai, Tue Le, Dung Nguyen Manh, Huy Phan Nhat, and Nghi D. Q. Bui. SWE-EVO: Benchmarking coding agents in long-horizon software evolution scenarios. *arXiv preprint arXiv:2512.18470v1*, 2025. URL https://arxiv.org/abs/2512.18470v1.

Harold Valdivia Garcia and Emad Shihab. Characterizing and predicting blocking bugs in open source projects. In *Proceedings of the 11th working conference on mining software repositories*, pp. 72–81, 2014. doi: 10.1145/2597073.2597099. URL https://doi.org/10.1145/2597073.2597099.

Sanidhya Vijayvargiya, Xuhui Zhou, Akhila Yerukola, Maarten Sap, and Graham Neubig. Ambig-SWE: Interactive agents to overcome underspecificity in software engineering. In *International Conference on Learning Representations*, volume 2026, pp. 146716–146737, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/ed4deb2ba9d4640fb6a000c688c2972e-Paper-Conference.pdf.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/a4b6ad6b48850c0c331d1259fc66a69c-Paper-Conference.pdf.

Hjalmar Wijk, Tao Lin, Joel Becker, Sami Jawhar, Neev Parikh, Thomas Broadley, Lawrence Chan, Michael Chen, Josh Clymer, Jai Dhyani, Elena Ericheva, Katharyn Garcia, Brian Goodrich, Nikola Jurkovic, Holden Karnofsky, Megan Kinniment, Aron Lajko, Seraphina Nix, Lucas Sato, William Saunders, Maksym Taran, Ben West, and Elizabeth Barnes. RE-Bench: Evaluating frontier AI R&D capabilities of language model agents against human experts. *arXiv preprint arXiv:2411.15114v2*, 2025. URL https://arxiv.org/abs/2411.15114v2.

xAI. Grok 4.1 Fast and agent tools API. https://x.ai/news/grok-4-1-fast, 2025. Accessed 2026-09-25.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying LLM-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024. URL https://arxiv.org/abs/2407.01489.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024a. doi: 10.52202/079017-1601. URL https://doi.org/10.52202/079017-1601.

John Yang, Carlos E Jimenez, Alex L Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R Narasimhan, et al. SWE-bench Multimodal: Do AI systems generalize to visual software domains? *arXiv preprint arXiv:2410.03859*, 2024b. URL https://arxiv.org/abs/2410.03859.

John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, Diyi Yang, and Ofir Press. ProgramBench: Can language models rebuild programs from scratch? *arXiv preprint arXiv:2605.03546*, 2026. URL https://arxiv.org/abs/2605.03546.

Z.AI. New released. https://docs.z.ai/release-notes/new-released, 2026. Z.AI Developer Documents; GLM-5.1 release entry dated April 7, 2026. Accessed: 2026-09-25.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Shulin Xin, Linhao Zhang, Qi Liu, Li Aoyan, Lu Chen, Xiaojian Zhong, et al. Multi-SWE-bench: A multilingual benchmark for issue resolving. *Advances in Neural Information Processing Systems*, 38, 2025. doi: 10.52202/085713-2111. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/5afa9cb1e917b898ad418216dc726fbd-Paper-Datasets_and_Benchmarks_Track.pdf.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous program improvement. *arXiv preprint arXiv:2404.05427*, 2024. URL https://arxiv.org/abs/2404.05427.

Zhaoxi Zhang, Yiming Xu, Jiahui Liang, Weikang Li, Xiaoshuai Chen, Liwei Qian, Xin Pei, Jizhou Huang, Rui Sun, and Yunfang Wu. RepoZero: Can LLMs generate a code repository from scratch? *arXiv preprint arXiv:2605.07122v3*, 2026. URL https://arxiv.org/abs/2605.07122v3.

Qixing Zhou, Jiacheng Zhang, Haiyang Wang, Rui Hao, Jiahe Wang, Minghao Han, Yuxue Yang, Shuzhe Wu, Feiyang Pan, Lue Fan, et al. FeatureBench: Benchmarking agentic coding for complex feature development. *arXiv preprint arXiv:2602.10975*, 2026. URL https://arxiv.org/abs/2602.10975.

Yuxuan Zhu, Tengjun Jin, Yada Pruksachatkun, Andy Zhang, Shu Liu, Sasha Cui, Sayash Kapoor, Shayne Longpre, Kevin Meng, Rebecca Weiss, et al. Establishing best practices in building rigorous agentic benchmarks. *Advances in Neural Information Processing Systems*, 38, 2025. doi: 10.52202/085713-5547. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/f316275b44ee2de533102913828a8107-Paper-Datasets_and_Benchmarks_Track.pdf.

APPENDIX CONTENTS

Section	Topic	Page
A	**Benchmark construction and validation**	**16**
A.1	Task schema and information visibility	16
A.2	Human validation	17
A.3	Input anonymization and long-horizon composition	18
A.4	Materialized task example	19
A.5	Per-task statistics	21
B	**Evaluation protocol and experimental settings**	**24**
B.1	Model and harness settings	24
B.2	Model metadata	25
B.3	Evaluation prompts	26
B.4	Evaluation adapters	27
B.5	Runtime configuration	29
C	**RQ1: Model, harness, and continuation performance**	**31**
C.1	Evaluation metrics	31
C.2	Pass@3 results	32
C.3	Per-task billed tokens	33
C.4	Resolve Rate by source	33
D	**RQ2: Planning, implementation, and testing**	**35**
D.1	Planning metrics	35
D.2	Extended development metrics	36
E	**RQ3: Context renewal and regression**	**38**
E.1	Context rounds, regression, and run coverage	38
E.1.3	Analytical compaction-trigger sensitivity	39
E.1.4	Trace scope	39
F	**Related work**	**40**
G	**Limitations**	**41**

A BENCHMARK CONSTRUCTION AND VALIDATION

A.1 TASK SCHEMA FOR LOOP ENGINEERING EVALUATION

Each task is materialized as a self contained directory for loop engineering evaluation, separating the work graph, presented requirements, executable obligations, and reference solution. This separation maintains a distinction between the evaluated loop's input contract and the hidden artifacts used to validate obligations and compute loop trace metrics.

```
task_example/
├── base/                          — codebase
│   ├── src/                       — varies across tasks
│   └── ...
├── gold_patches/                  — ground-truth solution
│   └── {module}.diff × N
├── requirements/                  — natural-language specs
│   └── {module}.yaml × N
├── tests/                         — fail-to-pass validation
│   └── ...                        — varies across tasks
├── Dockerfile
├── docker-compose.yaml
├── task.yaml                      — task metadata
├── solution.sh                    — oracle apply script
├── run-tests.sh                   — test entry point
├── module_dag.yaml                — detailed DAG with edge labels
├── slug_diff_map.json             — module-to-diff mapping
├── unit_dag.json                  — module dependency graph
└── gold-patch.diff                — concatenated solution
```

The `Dockerfile` and optional `docker-compose.yaml` pin a reproducible execution environment.

`run-tests.sh` is the checkpoint test entry point used by the evaluation runtime to adjudicate the current workspace state.

`solution.sh` and `gold-patch.diff` store the reference human solution used to validate executable obligations and anchor implementation comparisons.

`task.yaml` stores the dependency parents and the bindings between units and hidden standard tests. These fields are removed from the agent-facing manifest, which retains only each unit's requirement, file or symbol scope, and public acceptance criteria.

The `tests/` subtree holds source derived and supplemented acceptance tests indexed to the unit they adjudicate.

All three task sources use the same interface. At the start of a run, the agent receives the sanitized requirements for every unit, but not the graph, standard tests, or scoring state. The evaluator calls `run-tests.sh` at each checkpoint and records the state vector $s_i^{(\tau)}$ used in Appendix C.1. Tests become active when their units enter the ready frontier and remain active after the units are completed.

On-disk layout of a single LOOPSBENCH task

```
<repo>-<task-id>/                  self-contained task bundle
  Dockerfile                       pinned base image, system deps, lockfile
  docker-compose.yaml              tasks with multiple services
  run-tests.sh                     checkpoint test entry point for the evaluation runtime
  solution.sh                      driver that applies the reference solution
  gold-patch.diff                  reference human solution as a unified diff
  task.yaml                        evaluator metadata and DAG topology
  units/                           public requirement bundles presented during evaluation
    unit_001.yaml                  requirement, scope, public acceptance criteria
    unit_002.yaml
    ...
    unit_NNN.yaml
  tests/                           source-derived and supplemented test files
    unit_001/                      tests adjudicating unit 001
    unit_002/
    ...
    unit_NNN/
  base/                            entry-commit codebase the patch is applied to
```

Figure 5: Schema of a materialized LOOPSBENCH task.

Table 5: Agent-visible and evaluator-only information. Active hidden tests and checkpoint outcomes are not reported to the agent.

Evaluated-loop visible	Evaluator only
Global task instruction; unit requirements, scopes, and public acceptance criteria	Prerequisite sets, full DAG, ready frontier, and newly released obligation state
Base repository; project native tests; outputs of commands initiated by the evaluated loop	Standard tests and their unit bindings; reference patches, gold solution, and offline construction transcripts
Tests authored and run by the evaluated loop	Checkpoint outcomes, accumulated regression obligations, and scoring state

A.2 HUMAN VALIDATION OF MATERIALIZED TASKS

Human review complements the executable construction checks by assessing semantic agreement between the source requirements and the standard tests. The preceding validation stages screen for trivial tests, ancestor leakage, and environment-only failures; the manual audit focuses on semantic mismatches that can remain even when those checks pass.

Coverage. The exhaustive audit reviewed $1{,}475$ development units and $2{,}293$ existing dependency edges across 96 tasks, including all non-segmented tasks and all segmented PR Sequences with fewer than 80 pull requests. From each of the remaining 16 large PR Sequences, we sampled 5% of the nodes and edges, with 5–25 nodes and 5–40 edges per task. This adds 199 units and 291 edges, bringing the audit coverage to $1{,}674$ units and $2{,}584$ edges. Sampling each task separately ensures coverage of every large repository.

Protocol. Two annotators independently compared each item with the source artifact and its executable test results. They checked whether the instructions preserve the source acceptance criteria and whether the assertions test the intended behavior, using unit boundaries and dependency evidence to interpret each requirement. A semantic mismatch occurs when a test's acceptance condition differs from the intended requirement despite passing the executable checks in Section 2.5. The annotators resolved disagreements before confirming findings.

Audit coverage and findings. Table 6 summarizes the review coverage. Human review identified five semantic mismatches between tests and source requirements. All five identified mismatches were corrected. Semantic mismatches were the only type of error found after the preceding construction checks.

Table 6: Human-review coverage after executable construction checks.

Review stratum	Tasks	Units	Edges
Exhaustive review	96	1,475	2,293
Sampled review	16	199	291
Total	112	1,674	2,584

Unit counts refer to reviewed development units and their associated test suites, not individual test cases. Edge counts refer to reviewed existing edges.

Interpretation. The audit targets the semantic questions that executable checks alone cannot settle. Reviewing every unit in the largest PR Sequences would require substantially more manual effort, so we use sampling within each of those tasks while reviewing the other 96 tasks exhaustively. The few observed semantic mismatches provide evidence against widespread disagreement between the tests and the intended requirements in the reviewed material. This is evidence about audit findings, rather than a claim that every unreviewed unit is error-free.

A.3 Input Anonymization and Long-Horizon Composition

LoopsBench derives its tasks from public development artifacts, including PR Sequences, Course Labs, and Research Evolutions. Its evaluation target is the sustained composition of dependent requirements into a working repository over a long execution horizon.

Input Side Anonymization. During Task Instrumentation (Section 2.5), every unit requirement is rewritten to remove identifiable tokens that could let a model recognize the source from the prompt: repository names, course identifiers (e.g., MIT 6.824, Stanford CS144), paper titles, and author attributions. For PR Sequences, the same rewriting policy is applied at the segment (SEG) level, where original commit messages and PR titles often include the feature name, issue tracker ID, or merging author. After this step, the task prompt contains the acceptance specification, public interfaces, and unit requirements and scopes. The agent can run project-native and self-authored tests, but standard tests, their unit bindings, and checkpoint outcomes remain evaluator-only, as specified in Appendix A.1. This rewriting reduces direct source lookup cues in the instructions.

Composition across dependency chains. Pretraining provides programming knowledge and reusable local techniques that form part of an LLM's underlying capabilities. Familiarity with an algorithm, interface, or implementation fragment may help an individual step. Resolving a LoopsBench task additionally requires composing many such steps across the prerequisite DAG: implementing compatible changes, preserving earlier obligations as the repository evolves, and maintaining useful state across context renewal. The benchmark measures how agents apply and integrate these capabilities through environment interaction and sustained development.

Problem-only comparison. When given only the problem statements without access to the task environment or agent tools, all models evaluated in this setting obtain a Resolve Rate of 0%. With the environment and agent tools available, the strongest evaluated configuration reaches 25.00% Resolve Rate (Table 2). These observations characterize the difficulty of end-to-end completion under the evaluated settings and motivate measuring executable, sustained composition across the full dependency chain.

Appendix G discusses the residual possibility of overlap between public source artifacts and model training data.

A.4 Materialized Task Example for Loop Engineering Evaluation

This appendix presents one materialized task, making the schema in Section 2 and Appendix A.1 concrete. The example is `task_echarts_seg05`, a released PR Sequences task listed in Table 7. It connects source artifacts to unit requirements, dependency routing, and executable obligations. Task and PR identifiers are retained here for traceability; input-side anonymization is described in Appendix A.3.

Native artifact. The task comes from Apache ECharts and covers option types, line rendering, axis labels, server-side rendering, and localization. Two representative source changes are PR #15711, which introduces soft clipping for line gradients, and PR #15938, which fixes interpolation when the visible region lies entirely between two gradient stops. Their reference patches provide concrete evidence for a prerequisite relationship.

Preprocessing output. The released segment contains **43** development-unit nodes and **20** source-module groups. Requirements are stored as separate YAML descriptions, and `slug_diff_map.json` links them to per-PR reference patches. The two gradient changes are represented by `line-gradient-softclip.yaml` and `line-gradient-between-stops.yaml`.

Materialized task bundle. The bundle separates requirements, the work graph, tests, and reference changes as described in Appendix A.1. Selected paths from the actual on-disk layout are:

Task bundle layout

```
task_echarts_seg05/
  Dockerfile
  docker-compose.yaml
  run-tests.sh
  solution.sh
  task.yaml
  base/
  module_dag.yaml
  unit_dag.json
  slug_diff_map.json
  requirements/
    line-gradient-softclip.yaml
    line-gradient-between-stops.yaml
    ...
  gold_patches/
    15711.diff
    15938.diff
    ...
  tests/
    15711/...
    15938/...
    ...
```

The `Dockerfile` uses a Node 14 base image and copies the base repository and requirement files into the workspace.

Per unit requirement. The following normalized evaluator-side view combines the abridged requirement for PR #15938, its source-derived scope, and its DAG and test metadata. The `parents` and `tests` fields are evaluator-only, as specified in Appendix A.1:

Per unit requirement (excerpt)

```
unit_id: "15938"
requirement: |
  Correct the clipped line gradient when the
  visible segment lies entirely between two
  color stops. Interpolate the visible colors
  even when no stop lies inside the segment.
file_scope:
  - src/chart/line/LineView.ts
symbol_scope:
  - clipColorStops
  - getVisualGradient
acceptance:
  - between-stop segments retain the interpolated gradient
  - no incorrect end color or malformed clipped ramp
parents:
  ["15599", "15662", "15711", "15727",
   "15731", "15732", "15746", "15847"]
tests:
  - tests/15938/test/ut/spec/series/LineGradientBetweenStops.test.ts
```

The requirement specifies the expected gradient behavior as an acceptance contract. The agent infers integration constraints from its public inputs and workspace observations.

Recovered DAG. The released DAG contains **140** edges over the **43** units. Its longest directed path has **12** nodes and its largest topological layer has **20** nodes, matching the Depth and Width entries in Table 7. In the local example, the edge $u_{15711} \rightarrow u_{15938}$ is supported by the patches: the first introduces `clipColorStops`, and the second extends that function to interpolate colors at both visible boundaries when no original stop falls inside the interval. Readiness of u_{15938} requires completing all eight direct parents listed in the requirement box.

Trial verification. The construction checks in Section 2.5 distinguish the later boundary-case obligation from the earlier clipping behavior. For u_{15938}, discriminativeness requires at least one of its tests to fail after all ancestor reference patches have been applied, then all of its tests to pass after its own patch is added, with successor patches unapplied. Non-triviality and solvability additionally require failure on the starting codebase and success on the full reference implementation. During evaluation, previously activated tests continue to run so that later edits must preserve earlier functionality.

A.5 Task Pool Statistics for Loop Engineering Evaluation

The 112 LoopsBench tasks decompose into 29 PR Sequences (Infra and Data, Backend, Frontend), 57 Course Labs (Game Engines, Systems, Apps and ML), and 26 Research Evolutions (Analysis and Verification, Systems and Networking, Compiler with Database and ML). This source mix spans 9 domains and covers software work with different dependency structures, routing burdens, and verification obligations.

Tables 7 and 8 report per-task file and line counts (#Files, #LoC), development-unit counts (#Num), module and dependency-edge counts (#Mod, #Edges), dependency depth and width (Depth, Width), and average and maximum implementation size (Avg. LoC, Max LoC). The Depth column has a median of 4 across the 112 tasks. These statistics describe the work graph and implementation scope used to quantify routing burden and obligation retention pressure in loop engineering evaluation. The split into two tables is for typesetting, and the row sets are disjoint and cover the full task pool.

Table 7: Structural statistics for LoopsBench tasks, Part 1.

Repository			Units	Task structure			Implementation target		
Task	**#Files**	**#LoC**	**#Num**	**#Mod**	**#Edges**	**Depth**	**Avg. LoC**	**Max LoC**	**Width**
task_ClickHouse_seg07	3.3k	511k	817	30	2121	34	625	71.4k	404
task_NodeBB_seg01	1.7k	281k	138	17	640	27	2.0k	40.0k	28
task_NodeBB_seg03	4.1k	294k	382	22	130	10	769	35.5k	342
task_NodeBB_seg05	4.6k	129k	355	20	127	10	363	11.3k	310
task_TypeScript_seg01	14.4k	169k	114	18	0	1	1.5k	51.3k	114
task_afl_fuzzing	11	5142	5	5	4	2	1.0k	1400	4
task_alloy_formal_modeling	10	6457	4	4	4	4	1.6k	1845	1
task_angr_concolic_execution	14	5963	5	5	5	3	1.2k	1295	3
task_ansible_seg11	4.9k	43.9k	319	20	704	17	138	11.8k	134
task_ansible_seg13	5.5k	25.4k	111	16	59	5	229	6010	69
task_architecture_riscv_labs	420	3221	15	15	7	3	215	229	9
task_boogie_dafny	21	7025	3	3	2	3	2.3k	2811	1
task_calcite_query_optimization	9	6411	3	3	2	2	2.1k	2741	2
task_cocos2dx_navmesh_medium	4.5k	5204	14	14	12	4	372	1143	9
task_cocos2dx_physics2d_ medium	4.2k	4700	18	18	21	7	261	503	9
task_compiler	11	1815	27	27	31	8	67.2	72	6
task_compiler_cmm_mips	29	2678	9	9	7	4	298	1839	6
task_compiler_fdmj_llvm	89	1301	7	7	5	4	186	472	2
task_compiler_rustlike_rv32	97	2076	6	6	6	4	346	606	2
task_compiler_sysy_rust	21	1903	6	6	5	4	317	591	2
task_cpachecker	21	1596	3	10	2	3	532	561	1
task_cs61b_extra_java_bundle	218	2834	22	22	13	3	129	308	10
task_cvc_smt_solver	31	8013	3	3	2	3	2.7k	3065	1
task_datastructs_java	219	1974	15	15	4	2	132	330	11
task_db_nanodb	417	1519	9	9	8	3	169	774	4
task_db_query_optimizer_labs	70	1277	26	26	41	26	49.1	58	1
task_db_storage_index_labs	240	2395	10	10	11	7	240	1374	3
task_dbcompiler	21	1214	24	24	21	10	50.6	161	5
task_dcqcn_rdma_evolution	2.3k	31.2k	4	11	3	4	7.8k	15.7k	1
task_deep_learning_three_ assignments	84	1362	16	16	10	4	85.1	202	7
task_deep_rl_coursework	203	3282	19	19	23	6	173	811	6
task_dist_sys_raft_go	89	5861	4	4	5	3	1.5k	1608	2
task_distributed_fs_chfs	2.1k	2869	12	12	15	7	239	1195	3
task_django_seg01	5.8k	16.5k	209	16	476	21	78.8	1687	45

Continued on the next page

Table 7 (continued)

Repository			Units	Task structure			Implementation target		
Task	**#Files**	**#LoC**	**#Num**	**#Mod**	**#Edges**	**Depth**	**Avg. LoC**	**Max LoC**	**Width**
task_django_seg05	6.4k	15.9k	221	16	357	15	71.8	1690	108
task_django_seg11	6.8k	10.3k	165	17	213	12	62.3	718	80
task_doop_souffle_static_analysis	17	6342	5	5	4	3	1.3k	1458	2
task_ds_five_assignments	27	4911	5	5	0	1	982	1231	5
task_echarts_seg01	1.3k	19.6k	52	18	140	14	377	3903	15
task_echarts_seg04	1.4k	25.9k	53	20	153	13	488	12.6k	22
task_echarts_seg05	1.5k	13.7k	43	20	140	12	319	3271	20
task_echarts_seg07	1.5k	9758	65	10	223	13	150	2093	29
task_echarts_seg08	1.6k	27.3k	77	11	340	20	354	4742	32
task_echarts_seg09	1.6k	6694	46	11	128	12	146	1037	26
task_echarts_seg10	1.6k	9060	47	24	72	9	193	2229	30
task_echarts_seg11	1.6k	12.0k	62	38	215	17	193	1679	31
task_echarts_seg13	1.9k	5743	46	34	60	7	125	1509	25
task_evosuite_test_generation	17	8216	6	6	6	3	1.4k	1434	3
task_frama_c	21	4659	3	3	2	3	1.6k	2095	1
task_framework_seg04	1.4k	130k	383	18	766	22	339	15.6k	167
task_gem5_architecture_simulation	4	6004	4	4	3	2	1.5k	1661	3
task_godot_bootstrap_medium	13.8k	5020	14	14	15	8	359	1394	3
task_godot_navigation2d_medium	13.8k	5236	11	11	12	5	476	1261	7
task_graph_processing	6	5399	3	3	2	3	1.8k	1819	1
task_graphics_raster_2d	16	1305	12	12	17	4	109	529	7
task_graphics_scotty3d	390	1417	8	8	2	3	177	205	6

Table 8: Structural statistics for LOOPSBENCH tasks, Part 2.

Repository			Units	Task structure			Implementation target		
Task	**#Files**	**#LoC**	**#Num**	**#Mod**	**#Edges**	**Depth**	**Avg. LoC**	**Max LoC**	**Width**
task_hadoop_seg05	14.5k	395k	318	16	597	16	1.2k	103k	142
task_java_ds_coursework	96	3994	17	17	10	3	235	522	7
task_jenkins_seg02	11.4k	494k	84	11	167	16	5.9k	43.5k	40
task_jenkins_seg05	12.3k	63.4k	44	7	40	7	1.4k	13.1k	21
task_jpf_symbolic_execution	1.2k	229k	3	3	2	3	76.4k	126k	1
task_ligra_graph_evolution	57	14.8k	2	8	1	2	7.4k	7990	1
task_linux011_kernel_hacks	137	2687	13	13	15	6	207	724	5
task_llvm_alive_verification	556	32.8k	3	3	2	2	10.9k	27.0k	2
task_lsm_vector_store_four_stage	1.2k	1832	6	6	6	4	305	1832	2
task_mininet_network_simulation	10	6240	5	5	5	3	1.2k	1339	3
task_ml_four_assignments	42	2002	10	10	7	3	200	651	5
task_mlir_circuit_ir	10	5037	3	3	2	3	1.7k	2237	1
task_mlp_numpy_coursework	59	1865	7	7	6	3	266	290	4
task_mlsys_autodiff_pa1	9	1260	20	20	34	10	63	360	3
task_monogame_math2d_medium	2.1k	5002	12	12	17	4	417	1393	4
task_monogame_transform3d_medium	2.1k	4481	8	8	12	4	560	2318	2
task_mysql_sql_labs	48	1269	9	9	7	3	141	167	5
task_nandteris	256	1458	6	6	8	5	243	257	2

Continued on the next page

Table 8 (continued)

Repository			Units	Task structure			Implementation target		
Task	**#Files**	**#LoC**	**#Num**	**#Mod**	**#Edges**	**Depth**	**Avg. LoC**	**Max LoC**	**Width**
task_navidrome_seg02	365	52.1k	64	11	9	3	814	11.0k	57
task_navidrome_seg04	571	27.2k	56	12	16	5	485	13.1k	44
task_navidrome_seg05	628	50.1k	80	12	14	4	626	17.4k	67
task_navidrome_seg07	752	6724	44	12	4	3	153	1448	41
task_net_os_labs	54	2019	18	18	10	6	112	146	9
task_networks_proxy_rdt	48	1518	22	22	15	3	69	75	14
task_node_seg03	37.7k	62.6k	321	24	358	15	195	3925	210
task_ns3_network_simulation	3.1k	337k	2	2	1	2	168k	190k	1
task_numpy_dl_aics_style	176	1304	20	20	23	12	65.2	74	3
task_opengl_glfw_labs	1.4k	1256	8	8	7	8	157	274	1
task_os_c_fs_labs	28	1225	7	7	1	2	175	639	6
task_os_distributed_practicals	19	1514	5	5	0	1	303	349	5
task_os_labs_asm_kernel	780	14.3k	11	11	9	5	1.3k	3874	4
task_panda3d_collision_hard	6.7k	4813	14	14	14	5	344	903	6
task_panda3d_pgraph_medium	6.6k	3767	12	12	11	4	314	895	5
task_perses_program_reduction	21	5853	5	5	5	3	1.2k	1283	3
task_rails_seg09	4.8k	9203	122	16	90	7	75.4	1146	78
task_raylib_shapes2d_medium	1.4k	5083	10	10	18	5	508	1491	5
task_raylib_textures_hard	1.4k	6359	11	11	26	5	578	1548	5
task_rdma_ns3_evolution	2.3k	8835	3	12	2	3	2.9k	3241	1
task_riscv_cpu_six_labs	305	3426	6	6	6	3	571	720	4
task_riscv_cpu_sv	133	1872	12	12	17	6	156	449	5
task_riscv_cpu_verilog_three	56	1569	6	6	6	5	262	287	2
task_riscv_os_multiproject	619	4251	7	7	5	6	607	2006	2
task_sat_minisat_glucose	29	17.3k	2	9	1	2	8.7k	16.9k	1
task_security_labs	79	1642	25	25	23	6	65.7	71	7
task_sicp_py_scheme_sql	696	3132	22	22	25	13	142	698	5
task_sklearn_ml_toolchain	4	5252	3	3	3	3	1.8k	2445	1
task_soot_taint_analysis	10	5212	3	3	2	3	1.7k	1928	1
task_spring_course_mgmt_fullstack	19.8k	2837	10	10	15	4	284	338	3
task_sql_engine_myjql	44	1260	5	5	5	4	252	524	2
task_stream_mining	5	5245	5	5	5	3	1.0k	1241	2
task_systems_csapp_labs	101	1286	6	6	1	2	214	323	5
task_systems_four_labs	340	1910	10	10	5	3	191	623	6
task_tcp_course_stack	48	1402	2	2	1	2	701	1192	1
task_x86_os_four_labs	128	2229	9	9	11	6	248	600	2
task_xjqkl	26	1307	6	6	9	4	218	632	3
task_z3_fixedpoint_verification	11	6074	5	5	5	4	1.2k	1345	2

B Evaluation Protocol and Experimental Settings

B.1 Evaluation Settings for Model and Harness Configurations

Models and provider routes. RQ1 (E1) compares eight model checkpoints under a fixed Claude Code harness: Opus-4.7, GPT-5.5, GLM-5.1, DeepSeek-V4P, Gemini-3.1-Pro, Qwen3.6-Plus, Kimi-2.6, and Grok-4.1-FR. RQ1 (E2) compares agent harnesses by fixing the model to `azure/gpt-5.4-20260305` and varying Claude Code, Codex, GitHub Copilot, OpenHands, SWE-agent, and mini-SWE-agent. RQ2 analyzes the fixed-model harness runs with outer continuation from RQ1 (E2), matching the *w/ outer* setting in Table 2. RQ1 (E3) examines within family model changes while pairing each family with its corresponding agent harness for GPT, Claude Opus, and Qwen. RQ3 uses GPT-5.5 for Codex goal mode and the Ralph loop implemented with mini-SWE-agent, and Opus-4.7 for Claude Code goal mode and Claude dynamic workflows. Ralph uses outer continuation across fresh mini-SWE-agent invocations. Its GPT-5.5 configuration is distinct from RQ1 (E2)'s fixed GPT-5.4 harness comparison. Provider access is normalized through routes implemented in `long_horizon_bench/agents/`, including an Azure OpenAI deployment with AD token auth (`api-version 2025-04-01-preview`), the GitHub Copilot token exchange endpoint, a DashScope Anthropic compatible proxy, and direct Anthropic SDK access for Anthropic native models. This normalization separates provider transport from the loop engineering factors attributed to model choice and agent harness.

Decoding and evaluation input. Each agent harness inherits provider defaults unless its evaluation adapter records an explicit override. For the Anthropic SDK path used by Claude Code, we set `max_tokens=128,000` and leave temperature and top p at their SDK defaults. The SWE-agent LiteLLM call uses `temperature=0`. Top p is disabled because Anthropic rejects requests that set both fields. mini-SWE-agent forwards `model.model_kwargs.reasoning_effort=high` to providers that accept it. Agent harnesses receive the evaluation contract in Appendix B.3; no additional custom system prompt is supplied unless the evaluation adapter defines one.

Runtime budget and concurrency. The evaluation runtime imposes a per task wall clock budget $T_{\max}$ chosen per task ($\geq$2 h, capped at 24 h for the longest PR Sequence and Research Evolution tasks) and passes it to each agent harness as `timeout_sec`. For agent harnesses with their own iteration cap, we set the cap above the wall clock budget, making wall clock time the evaluation horizon. Claude Code SDK uses `max_iterations=10,000`, and mini-SWE-agent uses `-cost-limit 0` (uncapped) with `-exit-immediately`. Each bash tool call inside the container is independently capped at 600 s, separating local command stalls from sustained loop progress. Trials run concurrently in a `ThreadPoolExecutor`. Per trial credentials are passed as explicit kwargs through the task runner, keeping cross trial authorization races outside attribution to the evaluated agent harness.

B.2 Model Metadata for Fixed Loop Comparisons

For the within family scaling panel in Table 2, we record release date and reported parameter scale for each model evaluated under a fixed agent harness. The three families begin at different release points and follow different release cadences. The GPT family spans `gpt-4o` (May 2024) through `gpt-5.5` within the RQ1 evaluation window, the Claude family spans Sonnet-4 through Opus-4.7, and the Qwen family spans Qwen2.5-72B (open weight, 72 B parameters) through Qwen3.6-Plus (closed weight, undisclosed scale). Because the columns mix open and closed weight checkpoints, the metadata contextualizes fixed-harness comparisons rather than defining a model capability ranking. Resolve Rate monotonicity within each family is therefore interpreted as a generational delta under a fixed agent harness, not as a parameter count scaling law.

Table 9: Model metadata for RQ1 comparisons. Release dates are reported by month, including public preview releases; the Qwen3.6-Plus date follows its first dated official snapshot. Parameter counts separated by a slash denote total / activated parameters. "Not reported" means that the cited official source does not state the model's parameter scale.

Model	Family	Release	Params	Ref.
GPT-4o	GPT	2024-05	not reported	(OpenAI, 2024a;b)
GPT-5	GPT	2025-08	not reported	(OpenAI, 2025b)
GPT-5.2	GPT	2025-12	not reported	(OpenAI, 2025c)
GPT-5.4	GPT	2026-03	not reported	(OpenAI, 2026b)
GPT-5.5	GPT	2026-04	not reported	(OpenAI, 2026c)
Sonnet-4	Claude	2025-05	not reported	(Anthropic, 2025a)
Sonnet-4.5	Claude	2025-09	not reported	(Anthropic, 2025c)
Opus-4.5	Claude	2025-11	not reported	(Anthropic, 2025b)
Opus-4.7	Claude	2026-04	not reported	(Anthropic, 2026d)
Qwen2.5-72B	Qwen	2024-09	72 B	(Qwen Team, 2024a;b)
Qwen3-Max	Qwen	2025-09	$>$ 1 T	(Alibaba Cloud, 2025)
Qwen3.5-Plus	Qwen	2026-02	397 B / 17 B	(Alibaba Group, 2026)
Qwen3.6-Plus	Qwen	2026-04	not reported	(Alibaba Cloud, 2026)
Gemini-3.1-Pro	Gemini	2026-02	not reported	(Google DeepMind, 2026)
GLM-5.1	GLM	2026-04	not reported	(Z.AI, 2026)
DeepSeek-V4P	DeepSeek	2026-04	1.6 T / 49 B	(DeepSeek-AI, 2026)
Kimi-2.6	Moonshot	2026-04	1 T / 32 B	(Moonshot AI, 2026b;a)
Grok-4.1-FR	xAI	2025-11	not reported	(xAI, 2025)

B.3 Evaluation Contract for Loop Engineering

The evaluation contract supplied to each loop implementation is composed by `long_horizon_bench/harness/instruction_composer.py`. It combines task wording, planning requirements, and evaluation adapter addenda into the reproducible input observed by the evaluated loop.

For each evaluated loop, the input combines the task instruction, the *Benchmark Contract*, and a profile addendum selected by evaluation adapter class. The fixed-model runs shared by RQ1 (E2) and RQ2 replace the contract's optional planning paragraph with the *DAG Plan Requirement*; planning remains optional in the other runs. The Oracle adapter bypasses composition.

The literal prompt text below remains unchanged because it is the evaluated input. Its nonempty-patch requirement concerns trajectory submission and recording; it is not an additional functional scoring condition (Appendix B.4).

Benchmark Contract (always prepended; `instruction_composer._BENCHMARK_CONTRACT`)

```
You are running as a long-horizon autonomous engineering agent at `/workspace`.
This task spans many files and requirements, and is expected to take many
iterations.

Read every requirement under `/workspace/requirements/` (and any other explicit
requirement list referenced by the task) and only finish once every single one is
addressed. Cherry-picking or partial fixes will lead to failure.

## Optional planning artifact
If you believe the task is complex enough to benefit from a plan, you may write
`/workspace/agent_plans/plan.json` before making changes. This is optional. If you
create it, it must be a JSON object with `nodes` and `edges`. Each node must have
string `id` equal to a requirement slug from `/workspace/requirements/`, may
include
`label` and `layer`, and each edge must use string `from` and `to` that reference
those ids.

## Git workflow (required)
`/workspace` is initialized as a git repo. After finishing the implementation for
ONE requirement:
  1. `cd /workspace`
  2. `git add -A`
  3. `git diff --cached > requirement_patches/<slug>.diff`
  4. `git commit -m "impl: <slug>"`
Where `<slug>` is the requirement filename without the `.yaml` extension. The patch
file MUST be non-empty for the requirement to count as complete. Never write or
`touch` an empty `.diff` file.

You may run `git log`, `git diff`, `git show` at any time to inspect what prior
work has done -- only your own commits will be there; no reference solution is
accessible.

## Recommended workflow
1. Read the task carefully and survey `/workspace` so you understand the code you
   are changing.
2. As a development agent, when implementing a requirement you may write tests
   under `/workspace/agent_tests/` to safeguard correctness and future stability --
   write them if you think they help, otherwise skip.
3. Keep iterating until every requirement is implemented. Do not stop at a partial
   fix, a first passing check, or one successful edit.
4. Do not narrow the scope on your own. Missing a requirement is a task failure
   even if the changed tests pass.
5. You may revise the same files and rerun focused checks as many times as needed
   before finishing.
6. Your final summary must report requirement coverage counts, including at least
   the total number of requirements reviewed and how many were implemented and
   blocked.
```

Required DAG Plan (shared RQ1 E2 and RQ2 runs)

```
## Required planning artifact (DAG)
Before making any code changes you MUST write `/workspace/agent_plans/plan.json`.
It must be a JSON object with `nodes` and `edges` representing a directed acyclic
graph (DAG) over the requirement set. The DAG should reflect your view of which
requirements can be implemented in parallel and which must be serialized:

- `nodes`: an array. Each node is {"id": "<slug>", "label": "...", "layer": <int>}.
  `id` MUST exactly equal a requirement filename under `/workspace/requirements/`
  without the `.yaml` extension. Every requirement slug MUST appear as a node --
  no omissions.
- `edges`: an array. Each edge is {"from": "<slug>", "to": "<slug>"} meaning the
  `from` requirement must be implemented before `to` (a real ordering constraint,
  e.g. shared file/function, API surface change, or a logical prerequisite). The
  graph MUST be acyclic. Both endpoints MUST exist in `nodes`.
- `layer` is your topological-layer index (0 for sources). Two nodes in the same
  layer should be safely parallelizable -- no edge between them and no implicit
  conflict.

Implement requirements in an order consistent with the DAG. If you discover the
dependency structure is wrong while implementing, update `plan.json` first, then
continue. Do NOT skip writing this file.
```

Profile – `strong` (Claude Code)

```
Agent profile:
- Focus on completing the task directly.
- Do not spend tokens on basic shell tutorials or extended environment narration.
- Do not silently narrow the task scope, claim success early, or stop after partial
  requirement coverage.
- Use concise progress updates, keep modifying the workspace when needed, and
  validate the parts you change.
```

Profile – `structured_open_source` (SWE-agent, OpenHands)

```
Agent profile:
- Work step by step and keep actions grounded in the repository state.
- Prefer short, concrete progress updates and targeted validation.
- Avoid unnecessary tutorial-style shell explanation.
```

Profile – `legacy_verbose` (mini-SWE-agent)

```
Agent profile:
- You may use a slightly more explicit step-by-step style when it helps execution.
- Keep progress updates concrete and repository-specific.
- Prefer direct validation of changed behavior before finishing.
```

For Codex and GitHub Copilot, the evaluation adapter leaves the profile addendum empty. These loops receive the Benchmark Contract, with the required DAG plan replacement in the shared RQ1 (E2) and RQ2 runs.

B.4 Evaluation Adapter Details

The fixed-model comparison in RQ1 (E2) evaluates six coding agent harnesses through a uniform `BaseAgent` contract: Claude Code, Codex, GitHub Copilot, OpenHands, SWE-agent, and mini-SWE-agent. RQ2 uses their runs with outer continuation. The evaluation adapters normalize credential handling, model identifiers, and workspace conventions, so Table 2 can vary model choice and loop implementation while holding the task contract fixed.

Each run records the evaluation adapter boundary because it determines how an evaluated loop receives requirements, commits work, records plans, and returns artifacts used to compute loop trace metrics.

Workspace and git contract. Each evaluated loop executes inside `/workspace` in a separate Docker container for each task attempt, built from the task's `Dockerfile`. This provides every adapter with the same filesystem and execution boundary while isolating attempts from one another.

The directory is initialized as a git repository with three subtrees recorded by the evaluation runtime. `requirements/` contains per unit YAML manifests presented during evaluation, `agent_plans/` stores `plan.json`, and `requirement_patches/` stores one `<slug>.diff` per implemented unit derived from `git diff --cached`. The plan is required in the fixed-model runs shared by RQ1 (E2) and RQ2, and optional in the other runs (Appendix B.3).

A unit enters the recorded implementation set after its patch file is nonempty. This is a trajectory submission requirement: it determines whether an implementation artifact is recorded for the unit, not whether the unit is functionally complete. Functional unit completion and task Resolve Rate are determined by the hidden standard tests under the evaluation protocol. A missing or empty patch file does not independently turn a passing unit or task into a functional failure.

Checkpoint detection (`long_horizon_bench/harness/regression_harness.py`) uses `min_diff_lines=5` as the minimum diff growth for a snapshot trigger. The per unit threshold is derived from the smallest gold artifact currently on the ready frontier, making snapshot frequency follow recorded implementation progress rather than transcript length or turn count.

Evaluation adapters. *Claude Code* is evaluated through the Anthropic Agent SDK with a `ContainerWorkspaceBackend` that proxies the SDK bash and edit tools into the task container. Loop traces and per trial usage are written to `claude_sdk_trajectory.jsonl` and `claude_sdk_result.json`.

Codex uses a custom `model_provider` block, `lhb_azure` or `lhb_proxy`, injected into `config.toml`. This preserves the Codex invocation contract while routing model calls through the evaluation provider layer, keeping provider transport separate from loop behavior.

GitHub Copilot CLI is evaluated in noninteractive mode against a sidecar at `AZURE_SIDECAR_PORT=23337`, reached through the container default gateway IP in default compose setups that omit `host.docker.internal`.

OpenHands uses the fixed GPT-5.4 model in RQ1 (E2) and RQ2. Its adapter supplies the Benchmark Contract and the `structured_open_source` profile in Appendix B.3, including the required DAG plan instruction. It follows the same task-container workspace and requirement-patch recording contract described above. Outer continuation preserves that workspace across invocations, and RQ2 uses the latest recorded plan at the end of the complete attempt.

mini-SWE-agent is installed inside the container at `/opt/python` through `pip --prefix` with an explicit `PYTHONPATH` adjustment. Models prefixed with `copilot/` are rewritten to `openai/`, allowing LiteLLM to reach the OpenAI compatible proxy.

SWE-agent applies a compatibility patch that removes the `top_p` keyword. Anthropic rejects requests carrying both `temperature` and `top_p`, while the SWE-agent defaults set both fields.

Credential isolation. Earlier versions of the evaluation adapter wrote Azure AD tokens and Copilot keys into `os.environ` inside a context manager. Under the `ThreadPoolExecutor` used by the evaluation runtime, concurrent trials could overwrite shared tokens, and cleanup from one trial could clear a credential entry while another trial remained active. The current evaluation adapters pass credentials as explicit kwargs to `run_task` (`azure_ad_token_provider`, `copilot_api_key`, `anthropic_api_key`), preventing credential races from being attributed to the evaluated loop.

B.5 LOOP ENGINEERING EVALUATION RUNTIME CONFIGURATION

The LOOPSBENCH evaluation runtime is invoked via `lhb run --config config.yaml`, and the resulting run artifact records the loop parameters used for attribution. The evaluation runtime treats the evaluation adapter, loop budget, and residual handoff policy as explicit loop engineering factors for each run. All parameters can be set in the YAML configuration file or overridden by CLI flags. Table 10 summarizes the runtime parameters used to separate attribution across model choice, evaluation adapter, budget, and residual handoff policy while keeping the evaluated task contract fixed.

Table 10: Runtime parameters for `lhb run`.

Group / Parameter	Default	Description
Dataset		
`dataset_path`	`tasks/`	Path to the task directory tree.
`task_ids`	all	Task ID globs to include (repeatable).
`exclude_task_ids`	`[]`	Task ID globs to exclude (repeatable).
Evaluation adapter		
`agent`	`oracle`	Built in evaluation adapter name.
`agent_import_path`	–	Import path for a custom evaluation adapter class.
`model`	–	LLM in `provider/model` format.
`agent_kwargs`	`{}`	Key value pairs forwarded to the evaluation adapter.
Output		
`output_path`	`runs/`	Root directory for run artifacts.
`run_id`	timestamp	Unique identifier for this run.
`n_tasks`	all	Cap on number of tasks to evaluate.
Build		
`no_rebuild`	`false`	Skip rebuilding Docker images.
`cleanup`	`true`	Remove images after run completes.
Logging		
`log_level`	`info`	Python logging level.
`livestream`	`false`	Stream evaluated loop I/O in real time.
Concurrency		
`n_concurrent`	`4`	Tasks evaluated in parallel.
`n_attempts`	`1`	Attempts per task.
Timeout		
`global_timeout_multiplier`	`1.0`	Multiplier for all per task timeouts.
`global_agent_timeout_sec`	per task	Override loop budget (seconds).
`global_test_timeout_sec`	per task	Override test budget in seconds.

Per task timeouts. Each task declares `max_agent_timeout_sec` (default 1 800 s) and `max_test_timeout_sec` (default 3 600 s) in its `task.yaml`. The global overrides and multiplier allow uniform budget scaling without rewriting individual task files. Budget changes therefore remain attributable to run configuration rather than task materialization.

Residual handoff structure. The evaluation runtime executes an *outer loop* over evaluation adapter runs. Each round starts a fresh shell session while preserving the workspace from prior rounds, so residual handoff is measured as an observed loop engineering factor rather than treated as an implicit retry mechanism. The number of rounds is controlled by `outer_loop_count` (default 3); setting it to `0` or `unlimited` removes the cap, allowing repeated residual handoffs until the timeout budget is exhausted and making continuation policy explicit in the loop trace. Within each round, rate limit errors trigger retries governed by `rate_limit_retries_per_round` (default 5) with exponential backoff starting at `rate_limit_backoff_sec` (default 45 s), recording provider availability separately from loop progress.

Model and provider configuration. Provider credentials and environment variables are supplied via a separate YAML file (`model_config_path` or `--model-config`), which supports `env_files` (dotenv paths) and an explicit `env` map. This separation places provider transport outside the loop engineering factors attributed to model choice and evaluation adapter. Keys already set in the process environment are preserved unless `override_process_env: true` is selected, reducing cross run configuration drift.

Example configuration. The following minimal configuration specifies the dataset and execution policy while recording the evaluation adapter, loop budget, and residual handoff policy as explicit run metadata.

Run Configuration (`lhb.run.yaml`)

```yaml
dataset_path: tasks
output_path: runs

agent: mini-swe-agent
model: anthropic/claude-sonnet-4-20250514

n_concurrent: 4
n_attempts: 1

no_rebuild: false
cleanup: true
log_level: info
livestream: false

agent_kwargs:
  outer_loop_count: 3
  rate_limit_retries_per_round: 5
  rate_limit_backoff_sec: 45

global_timeout_multiplier: 1.0
```

Model Environment (`lhb.model.yaml`)

```yaml
env_files: []
override_process_env: false
env:
  ANTHROPIC_API_KEY: "sk-ant-..."
```

C RQ1: MODEL, HARNESS, AND CONTINUATION PERFORMANCE

C.1 METRICS FOR LOOP ENGINEERING EVALUATION

We formalize five evaluation metrics used to assess model and loop configurations on LOOPSBENCH. Together, they separate end state scoring from loop trace metrics for dependency progress, obligation retention, and resource use. Let $\mathcal{T}$ be the complete, fixed set of standard tests for a task, with tests indexed by i, and let evaluation checkpoints be indexed by $\tau \in \{1, \ldots, T\}$. Let $s_i^{(\tau)} \in \{0, 1\}$ denote the scored pass indicator: it is one when test i has been released and passes at checkpoint τ, and zero otherwise. An unreleased test receives no passing credit; this does not imply that it was executed and failed. Let O_τ denote the set of *regression obligations* active at checkpoint τ (tests previously released and passed at some earlier checkpoint). Each frozen snapshot is one checkpoint. The evaluator first reruns all previously activated tests, then activates and tests every newly ready unit on that same snapshot, updating unit completion after each stage until no further unit can be activated. Only the resulting state is recorded in $S = [s_i^{(\tau)}]$. This procedure also applies to the final snapshot before scoring. Newly passing tests become regression obligations only at subsequent checkpoints; activation stages within a snapshot are not separate checkpoints. Let $D_\tau \subseteq \mathcal{T}$ denote the set of tests released by checkpoint τ, and let $T_{\max}$ denote the wall clock time budget. The test pass rate at checkpoint τ uses all standard tests as its denominator:

$$\mathrm{TPR}(\tau) \;=\; \frac{\left|\{\, i \in D_\tau : s_i^{(\tau)} = 1 \,\}\right|}{|\mathcal{T}|}. \tag{1}$$

C.1.1 RESOLVE RATE (RR)

$$\mathrm{RR} \;=\; \mathbb{1}\!\left[\bigwedge_{i \in \mathcal{T}} s_i^{(T)} = 1\right]. \tag{2}$$

RR records whether every test in the complete task suite passes at task end, including all accumulated regression obligations. Passing only the released subset does not resolve a task if any tests remain unreleased. Benchmark RR is the mean of this indicator over all 112 tasks. It is the strict end state outcome metric, while the remaining metrics record dependency progress, obligation retention, and resource use along the recorded loop trace.

C.1.2 TEST PASS RATE (TPR)

$$\mathrm{TPR} \;=\; \frac{1}{|\mathcal{T}|} \sum_{i \in \mathcal{T}} s_i^{(T)}. \tag{3}$$

TPR records the fraction of all standard tests passing at task end. The denominator $|\mathcal{T}|$ is fixed for each task across checkpoints, models, and loop configurations; releasing further tests never changes it. Tests that remain unreleased receive zero passing credit. Benchmark TPR is the arithmetic mean of the per-task rates over all 112 tasks, with equal task weights; tests are not pooled across tasks. Unlike RR, it preserves partial progress and connects the final state to the intermediate loop trace recorded during execution.

C.1.3 DEPENDENCY DEPTH (DEPTH)

Depth measures dependency progress at the end of the first outer evaluation round, before residual continuation begins. Each test $i \in \mathcal{T}$ inherits its development unit's topological layer, $\mathrm{layer}(i) \in \{0, \ldots, L\}$, where $L = \max_{v \in V} \mathrm{layer}(v)$ counts edges on the longest prerequisite chain. The structural Depth in Tables 7 and 8 counts layers instead, and equals $L+1$. Let $\tau^\dagger$ be the last checkpoint of the first outer evaluation round, before any residual restart, and let $P_{\tau^\dagger} = \{i \in \mathcal{T} : s_i^{(\tau^\dagger)} = 1\}$. Let $\mathcal{F} \subseteq \{0, \ldots, L\}$ contain precisely those layers ℓ for which every test $i \in \mathcal{T}$ with $\mathrm{layer}(i) \le \ell$ belongs to $P_{\tau^\dagger}$. The per-task score is

$$d = \begin{cases} \max \mathcal{F}/L, & L > 0 \text{ and } \mathcal{F} \neq \varnothing, \\ 1, & L = 0 \text{ and } P_{\tau^\dagger} = \mathcal{T}, \\ 0, & \text{otherwise.} \end{cases} \tag{4}$$

Thus an incomplete root layer scores zero; an edgeless DAG ($L = 0$) scores one only when all task tests pass. Benchmark Depth is the arithmetic mean of d over all 112 tasks, with equal task weights. Depth records initial dependency progress, while RR records final completion after any residual continuation. High Depth with low RR can reflect unfinished later units, regressions, or both; the two aggregate scores alone do not distinguish these cases. Regression claims require observed pass-to-fail transitions in the checkpoint history, as defined below.

C.1.4 Regression Rate (Reg)

A *regression event* is a pass-to-fail transition of a previously passing obligation test. Checkpoints τ are snapshots emitted by the dual container watcher (Section 2.6). When the workspace diff meets the configured snapshot trigger (Appendix B.4), the evaluator runs the released test suite against the frozen snapshot in the evaluation container. It records per test pass or fail outcomes in `observation_history.jsonl` alongside the `newly_passed` and `newly_failed` deltas relative to the previous snapshot. For task j, let $O_{j,\tau}$ contain all tests that passed at a strictly earlier checkpoint, and let τ^- denote the preceding checkpoint at which test i was evaluated. Define the regression count and accumulated obligation exposure as

$$\begin{aligned} R_j &= \#\big\{(i,\tau) : i \in O_{j,\tau},\ s_{j,i}^{(\tau^-)}=1,\ s_{j,i}^{(\tau)}=0\big\}, \\ E_j &= \sum_{\tau} |O_{j,\tau}|. \end{aligned} \tag{5}$$

Table 3 pools numerators and denominators across all $N = 112$ tasks:

$$\mathrm{Reg} = \begin{cases} \dfrac{\sum_{j=1}^{N} R_j}{\sum_{j=1}^{N} E_j}, & \sum_{j=1}^{N} E_j > 0, \\ \text{N/A}, & \sum_{j=1}^{N} E_j = 0. \end{cases} \tag{6}$$

This is not an arithmetic mean of per-task rates: tasks are weighted by their obligation exposure. A task with $E_j = 0$ also has $R_j = 0$ and contributes neither numerator nor denominator. This includes tasks where no test ever passes, or tests first pass only at the final checkpoint. If total exposure is zero, retention is unobserved and Reg is N/A; with positive exposure and no regression events, Reg is zero. A test that remains failed contributes no additional pass-to-fail event until it passes and fails again, although it remains in the obligation set.

Reg should be interpreted alongside TPR and obligation exposure, with RR reporting final completion. Low RR does not imply little exposure: a task may satisfy many obligations yet remain unresolved because one test fails. Low Reg with little exposure provides limited evidence of retention. Because exposure counts test–checkpoint pairs, the pooled value also depends on the number of tests passed and subsequent checkpoints; it is not a task success probability.

C.1.5 Tokens (Tok)

Tok records total tokens consumed per task, summed across input and output over the full loop trace. This metric reports the resource cost of sustaining the evaluated loop over long horizon execution and contextualizes loop trace improvements against runtime expenditure.

C.2 Pass@3 Results Under Fixed Agent Harnesses

Pass@1 in the within-family comparison (Table 2) is based on one trial per task, model, and harness pairing. Table 11 reports pass@3 for the same pairings, using three independent attempts per task. Each attempt starts from the task's initial workspace in a separate Docker container; code and workspace state are not carried between attempts. A task counts as resolved under pass@3 if at least one of its three attempts resolves it. Outer continuation, when used, occurs within an attempt and preserves that attempt's workspace; it is distinct from the three independent attempts. The within-family ordering is preserved, with modest gains over pass@1.

Table 11: Pass@1 and pass@3 Resolve Rate for within-family comparisons with outer continuation.

Model	pass@1 (%)	pass@3 (%)
GPT (Codex)		
GPT-5.5	21.43	23.21
GPT-5.2	13.39	15.18
GPT-5	8.04	9.82
GPT-4o	3.57	4.46

Model	pass@1 (%)	pass@3 (%)
Claude (Claude Code)		
Opus-4.7	25.00	26.79
Opus-4.5	17.86	19.64
Sonnet-4.5	12.50	14.29
Sonnet-4	7.14	8.93

Model	pass@1 (%)	pass@3 (%)
Qwen (Qwen Code)		
Qwen3.6-Plus	9.82	11.61
Qwen3.5-Plus	8.04	9.82
Qwen3-Max	1.79	2.68
Qwen2.5-72B	0.00	0.89

C.3 Per Task Billed Tokens for RQ1

Tok reports mean billed tokens per task in millions under outer continuation, matched to the RQ1 configurations in Table 2.

Table 12: Mean billed tokens per task (millions) for the RQ1 configurations.

Configuration	Tok↓
GPT (Codex)	
GPT-5.5	8.36M
GPT-5.2	9.10M
GPT-5	8.92M
GPT-4o	4.51M
Claude (Claude Code)	
Opus-4.7	6.91M
Opus-4.5	5.38M
Sonnet-4.5	4.21M
Sonnet-4	3.76M
Qwen (Qwen Code)	
Qwen3.6-Plus	6.96M
Qwen3.5-Plus	6.88M
Qwen3-Max	5.71M
Qwen2.5-72B	5.02M

Configuration	Tok↓
Model *(fixed harness: Claude Code)*	
Opus-4.7	6.91M
GPT-5.5	7.18M
GLM-5.1	4.37M
DeepSeek-V4P	3.59M
Gemini-3.1-Pro	4.99M
Qwen3.6-Plus	4.25M
Kimi-2.6	2.02M
Grok-4.1-FR	1.83M
Harness *(fixed model:* `gpt-5.4`*)*	
Codex	7.94M
Claude Code	7.03M
GitHub Copilot	8.26M
OpenHands	6.75M
SWE-agent	7.18M
mini-swe-agent	8.86M

C.4 Per Source Resolve Rate for Model and Harness Configurations

The main experiments report aggregate Resolve Rate and loop trace metrics over the 112 LOOPS-BENCH tasks. This appendix reports the base RQ1 setting before outer continuation, matching the *w/o outer* RR column in Table 2. The three task sources represent distinct evidence regimes with different routing burden, state continuity, and regression obligation retention profiles. PR Sequences inherit source evidenced dependency DAGs from merged commit history, Course Labs inherit module level dependency structure from curriculum design, and Research Evolutions inherit coarser successor chains from citation graphs. Per source reporting separates these loop engineering patterns before aggregation. Table 13 reports per source Resolve Rate for the 14 configurations in the fixed-harness model sweep and fixed-model harness sweep of RQ1. This split places the aggregate comparison in source context and relates each source to the routing and obligation retention burdens faced by each loop implementation. The aggregate column reproduces the corresponding *w/o outer* number from Table 2 and equals $\mathrm{RR}_{\mathrm{agg}} = \frac{1}{112}\left(29 \cdot \mathrm{RR}_{\mathrm{PR}} + 57 \cdot \mathrm{RR}_{\mathrm{Course}} + 26 \cdot \mathrm{RR}_{\mathrm{Research}}\right)$ up to rounding.

Two findings hold uniformly across model and harness configurations and frame the interpretation of the per source columns.

Finding 1: PR Sequences are uniformly the most difficult source. Across all 14 model and harness configurations, PR Sequence Resolve Rate reaches at most 3.45%, and 13 of 14 configurations resolve no PR Sequence tasks. The PR Sequence pool is dominated by merge chains from actively maintained repositories, where each segment often depends on cross module invariants established several PRs earlier. When an early segment fails to preserve a prerequisite invariant, downstream

progress becomes governed by obligation retention across dependent edits. This pattern is consistent with the RQ3 observation that regression events remain present across all four loop profiles.

Finding 2: Course Labs account for most resolved tasks. For every model and harness configuration, Course Labs account for $\geq 70\%$ of the resolved task count. Course Labs have a curriculum imposed module decomposition that often makes the development order recoverable from task structure. Per source Resolve Rate on Course Labs varies most with model and harness configuration differences, and it tracks the aggregate Resolve Rate ranking most closely.

Research Evolutions fall between the two extremes. The moderate pool size (26 tasks) provides a more reliable per source estimate than earlier preview cuts, while the strongest model and harness configurations resolve a limited number of tasks and the weakest configurations resolve none. At this sample size, the per source spread is read together with the column level rate.

Table 13: Resolve Rate by task source before outer continuation.

Harness	Model	PR Seq. (n=29)	Course (n=57)	Research (n=26)	All (n=112)
Model sweep under the Claude Code harness					
Claude Code	Opus-4.7	3.45%	24.56%	15.38%	16.96%
Claude Code	GPT-5.5	0.00%	21.05%	11.54%	13.39%
Claude Code	GLM-5.1	0.00%	21.05%	11.54%	13.39%
Claude Code	DeepSeek-V4P	0.00%	19.30%	7.69%	11.61%
Claude Code	Gemini-3.1-Pro	0.00%	14.04%	7.69%	8.93%
Claude Code	Qwen3.6-Plus	0.00%	10.53%	3.85%	6.25%
Claude Code	Kimi-2.6	0.00%	5.26%	3.85%	3.57%
Claude Code	Grok-4.1-FR	0.00%	5.26%	0.00%	2.68%
Harness sweep under a fixed `gpt-5.4` *model*					
Codex	gpt-5.4	0.00%	21.05%	11.54%	13.39%
Claude Code	gpt-5.4	0.00%	19.30%	11.54%	12.50%
GitHub Copilot	gpt-5.4	0.00%	17.54%	7.69%	10.71%
OpenHands	gpt-5.4	0.00%	10.53%	3.85%	6.25%
SWE-agent	gpt-5.4	0.00%	8.77%	3.85%	5.36%
mini-swe-agent	gpt-5.4	0.00%	8.77%	0.00%	4.46%

These patterns shape the interpretation of aggregate LOOPSBENCH Resolve Rate. A single aggregate comparison combines Course Labs variation with the limited spread in PR Sequences, and the harness sweep gap in Table 2 is driven largely by Course Labs. PR Sequence gains are clearer in the per source columns than in aggregate deltas. Doubling PR Sequence RR moves the aggregate by less than one percentage point while materially changing the loop engineering interpretation of routing burden and obligation retention.

D RQ2: PLANNING, IMPLEMENTATION, AND TESTING

D.1 PLANNING LOOP TRACE METRICS FOR LOOP ENGINEERING EVALUATION

We formalize the four planning metrics used in RQ2 for loop engineering evaluation. They summarize dependency agreement and plan structure using the final plan DAG (the latest recorded plan at task end) and the source recovered human DAG (Table 3). The graph comparison therefore measures dependency understanding after execution, including plan revisions, rather than initial planning alone. Let $G_A = (V_A, E_A)$ denote this final plan DAG and $G_H = (V_H, E_H)$ the human development DAG recovered from source records. Let $V_\cap = V_A \cap V_H$ denote their shared units.

Plan availability and averaging. A task has planning scores only when its final recorded plan can be interpreted as a nonempty DAG. A missing plan, an empty node set, or a record that cannot be parsed as a DAG gives N/A for all four planning metrics and is excluded from their means; it is not assigned zero. In particular, mini-SWE-agent does not produce a plan DAG for some tasks, so its planning means cover only tasks with a usable recorded plan. A nonempty DAG with no edges remains a valid plan and is scored normally. Each planning metric is a task macro average: defined task scores receive equal weight, and the denominator is the number of tasks with a defined score for that metric. Layer ρ has the additional exclusions specified below, so coverage can differ across metrics and harnesses. If no task has a defined score, the mean is N/A. These exclusions concern planning observations only; all 112 tasks remain in the functional evaluation.

D.1.1 EDGE F1

For two edge sets A and B, define

$$\mathrm{F}_1(A, B) = \begin{cases} 1, & A = B = \varnothing, \\ \dfrac{2|A \cap B|}{|A| + |B|}, & \text{otherwise.} \end{cases} \tag{7}$$

$$\text{Edge F1} \;=\; \mathrm{F}_1(E_A, E_H). \tag{8}$$

Both empty edge sets score one; exactly one empty set, or two nonempty disjoint sets, score zero. Edge F1 measures exact recovery of direct prerequisite edges. Low precision indicates extra direct edges relative to the recovered DAG; low recall indicates missing reference edges. Neither alone establishes an invalid prerequisite ordering: an extra edge may express an existing transitive dependency, and a missing direct edge may still be represented by an indirect path. For example, adding $A \to C$ to a reference chain $A \to B \to C$ preserves prerequisite reachability but reduces Edge F1 to 0.8. Edge F1 and Layer ρ are computed against the source-evidenced DAG validated in Appendix A.2. Because this DAG is conservative, neither metric measures agreement with a complete semantic dependency graph.

D.1.2 LAYER ρ

Let $\ell_A(v)$ and $\ell_H(v)$ denote the topological layer index of unit v in G_A and G_H. Layer 0 contains prerequisite free nodes, and later layers depend on earlier layers.

$$\begin{aligned} \text{Layer } \rho \;=\; \mathrm{Spearman}\big(&\{\ell_A(v)\}_{v \in V_\cap}, \\ &\{\ell_H(v)\}_{v \in V_\cap}\big). \end{aligned} \tag{9}$$

If $|V_\cap| < 2$ or either layer array is constant, Layer ρ is undefined and reported as N/A; that task is excluded only from this metric's mean and denominator, rather than assigned zero. Otherwise, Spearman correlation uses average ranks for ties. Layer ρ records whether the plan places units at similar development *stages* as the human reference, independent of exact edge structure. Low values indicate different stage assignments in the recorded plan. Evaluator readiness remains determined by the recovered DAG and checkpoint outcomes, rather than by the agent's planned layers.

D.1.3 CRITICAL PATH RATIO

Let $\mathrm{CP}(G)$ denote the length (in nodes) of the longest directed path in G.

$$\mathrm{CPR}(G) \;=\; \frac{\mathrm{CP}(G)}{|V|}. \tag{10}$$

Values near 1 correspond to a nearly linear recorded plan; lower values indicate a shorter critical path relative to the number of planned units. Whether a plan adds unnecessary ordering constraints requires comparison with the recovered DAG. CPR alone does not establish that the remaining branches are valid or executed concurrently.

D.1.4 WIDTH RATIO

Let $w(G) = \max_\ell |\{v \in V : \ell(v) = \ell\}|$ denote the maximum topological layer width of G, using the layer indices defined above.

$$\mathrm{WR} \;=\; \frac{w(G_A)}{w(G_H)}. \tag{11}$$

WR compares the largest topological layer in the recorded plan with that in the recovered DAG. Values below or above 1 indicate a smaller or larger maximum layer width, respectively. A value close to 1 indicates similar maximum widths, but does not establish agreement on dependencies or correctness of the plan. WR is a graph statistic: it does not measure the time-varying ready frontier, actual execution concurrency, or regression risk.

D.2 EXTENDED RQ2 LOOP TRACE METRICS

Loop trace metric definitions for Table 3. Table 3 uses the fixed GPT-5.4 harness runs with outer continuation from RQ1 (E2), corresponding to the *w/ outer* setting in Table 2. Trace metrics cover the initial invocation and its outer continuations; the final plan is taken at the end of the complete attempt. *Planning fidelity.* Planning metrics use the final plan recorded at task end, with the boundary and averaging conventions in Appendix D.1. Tasks without a usable nonempty plan DAG have N/A planning scores and are excluded from the corresponding means; this includes the mini-SWE-agent tasks for which no plan DAG was produced. **Edge F1** = F1 between the final plan DAG edge set and the human development DAG edge set. **Layer** ρ = Spearman correlation between topological layer indices in the final plan and those in the human development DAG. **CPR** = critical path length of G_A divided by $|V_A|$ (direct value, unnormalized to gold). **WR** = maximum topological layer width of G_A divided by that of G_H, as defined in Appendix D.1.4. It summarizes plan structure rather than observed execution concurrency.

Implementation metrics are computed on units resolved by each evaluated configuration, so the unit subsets may differ across configurations. **PatchLen** = patch lines per matched unit divided by reference patch lines. **Jacc** = token-level Jaccard distance between candidate and reference patches; larger values indicate less token overlap. The reported values are distances, not overlap fractions. A task with no resolved units has no implementation comparison: its PatchLen and Jacc scores are N/A and it is excluded from these means, rather than assigned zero. More generally, a per-task implementation score is defined only when at least one resolved unit has a matched patch pair for which that metric is defined. Reported values are arithmetic means of defined per-task scores, with equal task weights; each denominator is the number of tasks with a defined score for that metric. If no task qualifies, the aggregate is N/A. These exclusions do not remove tasks from RR, TPR, or F2P. Neither implementation metric alone measures code quality.

Testing metrics summarize verification work authored during the recorded loop trace. Each authored test is counted once, using its final recorded version; earlier versions of a modified test do not contribute additional tests to either #T or the F2P numerator and denominator. **#T** = number of distinct tests authored in the recorded loop trace per task, reported as the mean over all 112 tasks. **F2P** = the per-task fraction of these final test versions that fail on the initial state and pass on the gold solution. A task with no authored tests receives F2P $= 0$ and remains in the benchmark average. Table 3 reports the arithmetic mean of per-task F2P scores over all 112 tasks, with equal task weights. These endpoints assess discrimination between the initial and reference states, rather than the fraction of authored tests passing on the agent's submitted implementation. Reg uses the separate convention in Appendix C.1.4: it pools regression counts and obligation exposure across tasks.

Source of additional patch lines. On resolved units, the reported PatchLen ratios range from 1.58 to 2.54, with token-level Jaccard distances of 0.62–0.76. Recorded diffs include style changes, extensive comments, and rewrites of existing code blocks, which contribute additional patch lines. These observations describe differences from the reference implementation; they do not establish that

the added lines are unnecessary or that incomplete state retention caused them. The reference patch is not the only valid implementation.

E RQ3: Context Renewal and Regression

E.1 Context Round and Regression Metrics

E.1.1 Trace Definitions

RQ3 counts both outer invocations and compaction intervals as context-budget rounds, extracted directly from execution trajectories. For trial i, let K_i be the number of outer invocations and let C_i be the number of additional context segments identified from compaction events in the recorded trajectory. Each segment opens a new maximal context budget. We report

$$\text{Rounds} = \frac{1}{N_{\text{trace}}} \sum_{i=1}^{N_{\text{trace}}} (K_i + C_i). \tag{12}$$

An outer invocation opens an active context and is counted in K_i; subsequent invocations restart it. An internal compaction interval is counted in C_i. Codex uses the source-level auto-compaction limit, which defaults to 0.9 of the configured context window. Claude Code uses its configured context budget. Ralph repeatedly invokes mini-SWE-agent against a persistent workspace. For dynamic workflows, K_i and C_i count only the top-level coordinating session and its outer restarts; worker contexts are excluded, as described in Appendix E.1.4.

The regression watcher writes `observation_history.jsonl`. Let N_{reg} be the number of runs with regression histories and N_{trace} the number with context trajectories used for Rounds. Both equal 112 for every reported configuration. Let $F_{i,s}$ be the set of previously passing standard tests that newly fail at triggered snapshot s. Regression snapshots per run are

$$\text{Reg/run} = \frac{1}{N_{\text{reg}}} \sum_{i=1}^{N_{\text{reg}}} \sum_{s} \mathbb{I}[F_{i,s} \neq \varnothing]. \tag{13}$$

Each snapshot contributes at most one event, regardless of how many tests regress. Unlike RQ2's Reg, this metric averages snapshot-level occurrences per run rather than normalizing test-level transitions by the accumulated obligation exposure across checkpoints. Regression histories record changes in the shared workspace, including changes made by workers. Resolve Rate uses the benchmark outcome for every scheduled task:

$$\text{RR} = \frac{1}{N_{\text{task}}} \sum_{i=1}^{N_{\text{task}}} \mathbb{I}[\text{trial } i \text{ is resolved}]. \tag{14}$$

E.1.2 Run Coverage

All four configurations—Codex goal mode, Claude Code goal mode, Claude dynamic workflows, and Ralph—are evaluated on the same full release of 112 tasks. Every task has both a context trajectory for Rounds and a regression history for Reg/run, so $N_{\text{task}} = N_{\text{trace}} = N_{\text{reg}} = 112$ for each configuration. All three metrics therefore cover the same complete task set. Dynamic workflow Rounds uses the coordinator trajectory for each task, retaining the scope defined above. The resolved-task counts are 23, 20, 27, and 9, respectively, corresponding to RR values of 20.54%, 17.86%, 24.11%, and 8.04%. For the dynamic workflow run, all 112 regression histories are available and contain 42 triggered snapshots with newly failing tests, giving Reg/run $= 42/112 = 0.375$, reported as 0.38.

E.1.3 Analytical Compaction-Trigger Sensitivity

The non-default round counts in Table 4 come from an analytical sensitivity calculation, not from additional agent runs. Let D be the reported mean context-budget round count and K the mean number of outer invocations. At trigger multiplier m, we compute $D(m) = K + (D - K + 1)/m - 1$, keeping total context use fixed and varying only the interval length. We evaluate $m \in \{0.75, 1.00, 1.25\}$ for each profile. The Shorter and Longer columns correspond to $m = 0.75$ and $m = 1.25$, respectively. For dynamic workflows, D and K refer only to the coordinating session. The resulting values describe sensitivity in round count and do not predict changes in Regression Rate or Resolve Rate.

Under this fixed-usage assumption, earlier triggers increase the round count for every profile. Establishing a causal effect would require rerunning matched model–task pairs at different thresholds and comparing compaction frequency, post-event regressions, and final resolution. In the absence of those runs, we treat the relationship between renewal and regression as correlational.

E.1.4 Trace Scope

The dynamic workflow archive records the coordinating Claude Code session and outer restarts. It does not retain a uniform event stream for every worker process. The reported Rounds value of 97.96 is extracted directly from the coordinator trajectories and excludes worker contexts. The analytical sensitivity calculation uses the same scope. Worker completion state remains visible through repository changes, requirement patches, and regression snapshots.

Codex goal mode and Ralph use GPT-5.5, while Claude Code goal mode and Claude dynamic workflows use Opus-4.7. Ralph uses mini-SWE-agent with outer continuation, distinct from the fixed GPT-5.4 harness comparison in RQ1 (E2). These values characterize observed execution profiles; the full table is not a model-controlled causal comparison. RR, Rounds, and Reg/run each cover all 112 tasks for every configuration. Regression histories record changes in the shared workspace, while dynamic workflow context counts retain the coordinator-only scope. The sensitivity calculation uses fixed per-profile outer-invocation and context-round summaries and applies the analytical rescaling above; it does not run additional agent experiments.

F Related Work for Loop Engineering Evaluation

Table 14 expands Section 4 and lists sources for Table 1, grouped by their evaluation evidence and relation to loop engineering.

Table 14: Prior work related to loop engineering evaluation.

Cluster	References (representative + extensions)
Repository-level software engineering. Cover issue resolution, feature implementation, and related coding tasks and training environments, including test generation and fixes requested in code review.	Jimenez et al. (2024); Zan et al. (2025); Yang et al. (2024b); Deng et al. (2025); OpenAI (2024c); Miserendino et al. (2025); Pan et al. (2024); Sonwane et al. (2026)
SWE-bench audits. Document contamination and overfitting risks in verified issue resolution variants.	OpenAI (2026d); Zhu et al. (2025)
Feature and evolution. Evaluate features, refactoring, and evolution, including partial completion and preservation checks.	Zhou et al. (2026); Chen et al. (2025); Thai et al. (2025); Gautam et al. (2025); Feng et al. (2026)
Machine learning and research engineering. MLE-bench evaluates Kaggle competition performance against private-leaderboard medal thresholds; RE-Bench compares agents with human experts on open-ended AI research and development tasks.	Chan et al. (2024); Wijk et al. (2025)
Code optimization. Measure agentic optimization of existing code using performance workloads and task-specific performance objectives.	Sehgal et al. (2026)
Repository generation and reproduction. NL2Repo-Bench evaluates repository generation from natural-language specifications. ProgramBench provides executable access and documentation; RepoZero provides API specifications and examples.	Ding et al. (2025); Yang et al. (2026); Zhang et al. (2026)
Interaction and trace benchmarks. Examine task duration, tool interaction, and development workflows. LoopsBench connects traces to unit dependencies and preservation of newly completed functionality.	Kwa et al. (2026); Siegel et al. (2024); Li et al. (2024); Merrill et al. (2026); Edwards et al. (2026); Chen et al. (2026); Vijayvargiya et al. (2026)
Open-source coding agents. Implement repository navigation, editing, tool use, and repair workflows. LoopsBench compares harnesses using shared tasks and scoring rules.	Yang et al. (2024a); Wang et al. (2025); Zhang et al. (2024); Xia et al. (2024)
Industrial coding agents. Support development through terminal, editor, and cloud environments, motivating evaluation across different execution interfaces.	Anthropic (2025d); OpenAI (2025a); GitHub (2026); Anysphere (2026); Cognition (2026)
Persistent goals and worker orchestration. Sustain objectives and delegate work. LoopsBench relates execution profiles to completion, context renewal, and retained functionality.	OpenAI (2026a); Anthropic (2026a;c;b)
External continuation loops. Ralph repeatedly invokes an agent against a persistent workspace. LoopsBench tracks progress and regressions across invocations.	Huntley (2025; 2026)
Software engineering prerequisites. Study change impact, blocking dependencies, and release planning, providing source concepts that LoopsBench converts into executable obligations and loop trace metrics.	Lehnert (2011); Valdivia Garcia & Shihab (2014); Ruhe & Saliu (2005)

G LIMITATIONS

1. **Dependency structure.** The recovered DAG is a lower bound on prerequisite structure rather than a complete causal graph of development. Edge admission uses four source evidenced patterns and a hot file denylist. Missing prerequisite edges may cause dependency depth to be underestimated. Dependencies expressed only through configuration, data formats, build systems, or cross-service behavior may be missed. Unit boundaries and reference patches are also not unique, and the fixed-DAG formulation does not cover exploratory work in which earlier requirements are replaced. We therefore interpret the planning metrics relative to the recovered graph rather than to an assumed complete dependency structure. Appendix A.2 describes our manual review of semantic agreement between source requirements and tests.
2. **Task coverage.** The three pipelines admit tasks when commit history, curriculum decomposition, or citation closure yields verifiable evidence, narrowing the current pool to sources with auditable prerequisite evidence. Although the release includes some frontend and hardware-related tasks, it provides limited coverage of mobile development, frontend-intensive projects, and integration with physical hardware.
3. **Correctness and feedback.** Reported correctness metrics are tied to the released test suite, including TPR, Resolve Rate, and Regression Rate, and therefore measure executable obligations rather than full semantic equivalence. The construction checks verify behavior on the initial, ancestor-only, unit-complete, and full-reference states, but cannot fully certify semantic agreement between requirements and tests. Human review therefore targets this remaining gap and identified five semantic mismatches, all of which were corrected (Appendix A.2). Review covers all 112 tasks, with exhaustive review of 96 and sampling within the 16 largest PR Sequences because of annotation effort and cost. The small number of observed mismatches supports the view that semantic test errors are uncommon in the reviewed material, while incomplete coverage leaves some uncertainty about unreviewed units. Standard tests and checkpoint results are hidden, so regressions are measured under more limited feedback than a visible continuous-integration system would provide.
4. **Construction bias.** Benchmark construction with Claude Code is performed offline, after which the resulting tasks are fixed and shared by all evaluated agents. Source evidence, executable checks, and manual review support the validity of these tasks, but cannot eliminate a possible preference for Claude-like instructions, decompositions, environments, or tests. An independent reconstruction of a task subset is still needed to measure this effect.
5. **Public-source overlap.** We cannot fully rule out overlap between public source material and model training data. To reduce reliance on direct source recall, we remove explicit source identifiers from the instructions and evaluate executable behavior across the task's dependency structure, including preservation of previously completed requirements. Pretraining may supply useful programming knowledge or local implementation fragments, but recalling such fragments does not satisfy the complete task contract. In the problem-only setting without the task environment or agent tools, all evaluated models obtain 0% Resolve Rate (Appendix A.3). These design constraints and observations support interpreting the results as a measure of sustained implementation and composition: prior familiarity may help individual steps, while task resolution still requires integrating them into a working repository that satisfies all test obligations.
6. **Trace observability.** LOOPSBENCH records coding agent loop execution through traces made accessible by current loop implementations and evaluation adapters. Internal events outside these interfaces cannot be measured directly.
7. **Evolving loop implementations.** As goal modes, dynamic workflows, compaction policies, and external loop implementations evolve, LOOPSBENCH provides a loop engineering evaluation interface with replaceable adapters and version pinned run metadata.
8. **Cost and reproducibility.** Running the full benchmark is expensive, and proprietary model updates may change the results. The benchmark and harness can thus be inspected and rerun, although closed-source outputs may not reproduce exactly.
9. **Scope of evaluation.** Finally, LOOPSBENCH evaluates agents under a specific executable contract; it is not a general measure of software-engineering ability or deployment readiness. Agent-generated code still requires human review, security testing, and license checks.