HORIZONFLOW: Towards Diagnosable Long-horizon Evaluation for Coding Agents*

Anonymous ACL submission

Abstract

LLM-based coding agents are increasingly deployed on long-horizon software development, where later changes must build on earlier ones across many dependent steps. Existing benchmarks package work as isolated issues and score only terminal outcomes, offering limited insight into whether agents sustain prerequisite states and avoid regressions as the horizon grows. We introduce HORIZONFLOW, a long-horizon, process-oriented benchmark for dependency-structured coding. HORIZONFLOW comprises 112 tasks recovered from authentic, real-world sources spanning 8 programming languages and 9 domains, with a median dependency depth far above single-issue benchmarks. Each task is a dependency DAG whose nodes are separately testable development units and whose edges encode source-evidenced prerequisites. A flow-aware harness streams tests along the ready frontier and retains completed nodes as persistent regression obligations, yielding intermediate trajectories rather than a single terminal verdict. Using HORIZONFLOW we evaluate frontier LLMs and widely used coding agents. Even the strongest configuration resolves only about a quarter of tasks, agent trajectories diverge structurally from the human development order, and regressions concentrate around context compaction events as the horizon lengthens. HORIZONFLOW surfaces intermediate, dependency-aware signals that show where long-horizon agents break down, and we release the full task suite for community use.[1]

1 Introduction

Coding agents such as Claude Code (Anthropic, 2025b), GitHub Copilot (GitHub, 2026b), Cursor (Anysphere, 2024), and Windsurf (Codeium, 2025) are increasingly embedded in continuous development, where each change builds on prior ones rather than resolving an isolated issue. In this long-horizon setting, the core challenge is not implementing many independent requirements, but honoring prior commitments, building later work upon them, and avoiding regressions as the codebase evolves (Carlini, 2026).

Limitations of Existing Benchmarks. Existing coding benchmarks have increasingly moved beyond isolated issue resolution. SWE-bench and its variants broaden repository-level evaluation along freshness, language coverage, modality, and patch scale (Jimenez et al., 2023a; Zan et al., 2025; Yang et al., 2024c; Liu et al., 2025; Deng et al., 2025), while feature- and evolution-level benchmarks further enlarge the requested change (Zhou et al., 2026; Chen et al., 2025; Thai et al., 2025). Broader long-horizon benchmarks, including repository generation and process-oriented tasks, extend the time horizon even further (Kwa et al., 2025; Siegel et al., 2024; Merrill et al., 2026; Edwards et al., 2025; Ding et al., 2025; Feng et al., 2026). However, these benchmarks still largely rely on final task-level test outcomes over issues, features, coarse milestones, or whole-repository specifications. This evaluation style obscures whether an agent preserves prerequisites, accumulates obligations, avoids late regressions, or loses state as the context grows. Diagnostic long-horizon evaluation should therefore be defined not only by task scale, but by a harness that exposes process-level signals: streaming tests along dependency structure, retaining completed units as regression obligations, and relating agent trajectories to source-grounded human development order.

HORIZONFLOW: Towards Diagnostic Flow-Aware Testing. HORIZONFLOW (Figure 1) accordingly recovers each task as a dependency DAG whose nodes are source grounded development units and whose edges encode prerequisite rela-

*Dataset available at https://dataverse.harvard.edu/previewurl.xhtml?token=909b6ea4-4306-430e-95df-8b8ceaac9204.

[1] https://dataverse.harvard.edu/previewurl.xhtml?token=909b6ea4-4306-430e-95df-8b8ceaac9204

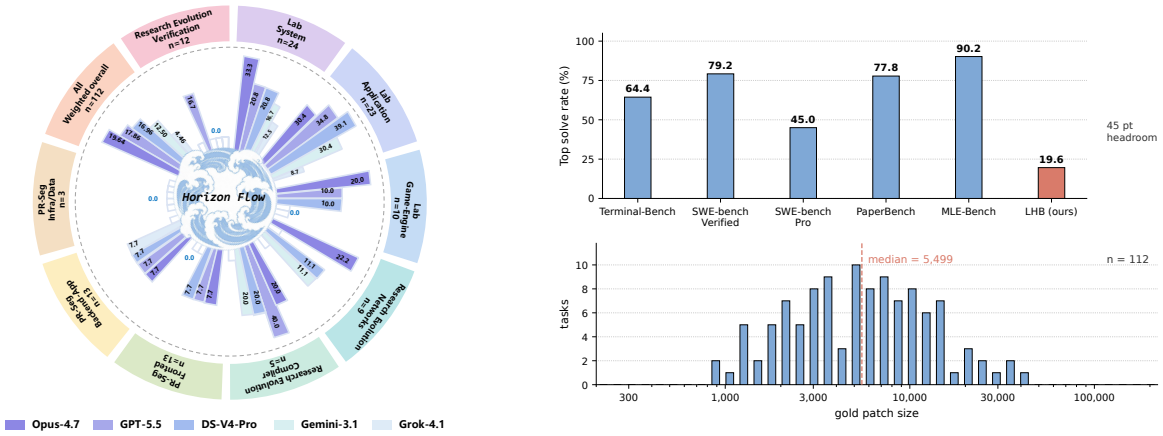

Figure 1: **HORIZONFLOW at a glance.** *Left:* per-category solve rate of flagship models under Claude Code on HORIZONFLOW. *Top right:* SOTA on popular code benchmarks as of 2026-05-20; HORIZONFLOW leaves substantial headroom even against agentic SWE-bench leaders. *Bottom right:* per-task gold-patch size over the 112 HORIZONFLOW tasks.

tions. The graph captures prerequisite commitments, makes intermediate units separately testable, and reconstructs a human development order from the underlying source records. The harness streams obligations along this graph: it reveals tests only for nodes on the ready frontier, keeps completed nodes active as regression obligations, and enforces prerequisite validity without prescribing a total execution order.

HORIZONFLOW contains 112 dependency-structured tasks from three authentic, real-world sources (29 PR Sequences, 57 Course Labs, and 26 Research Evolutions), spanning 9 domains and 8 programming languages, with a median dependency depth of 6 and over 5,300 development units. Table 1 situates HORIZONFLOW against existing coding benchmarks as the sole benchmark to pair explicit unit dependency DAGs with a flow-aware harness.

Take-Aways. Using HORIZONFLOW we evaluate frontier LLMs and widely used coding agents, and our analysis surfaces three key findings: ❶ **Long-horizon tasks remain far from solved.** Even the strongest model-scaffold pair resolves only about a quarter of the benchmark, indicating that long-horizon execution is still well outside the reach of current coding agents. ❷ **Agent behavior diverges from human development across plan, code, and test.** Agent plans drop pull request level dependencies relative to the human DAG, patches inflate beyond the gold change set, and the test suite stays sparse, so earlier obligations are left unguarded as the horizon extends. ❸ **Current harnesses sustain execution but routing and memory remain weak.** Outer loop scaffolds already keep the agent running across long horizons, yet task routing across handoffs and state retention across in context compaction are where regressions concentrate.

Contributions. Our contributions are:

1. **A graph-structured evaluation contract for long-horizon coding.** The HORIZONFLOW harness streams tests over executable dependency DAGs while retaining completed nodes as regression obligations.
2. **A scalable pipeline for structured, long-horizon coding tasks.** HORIZONFLOW automates task collection, selection, materialization, and dependency recovery across PR Sequences, Research Evolutions, and Course Labs, covering real software engineering, research, and university lab work under one uniform schema.
3. **A benchmark recovered from real, long-horizon development records.** HORIZONFLOW provides 112 tasks from Course Labs, PR Sequences, and Research Evolutions, with prerequisite structure recovered from source evidence rather than fabricated decompositions.
4. **A fine-grained empirical comparison of long-horizon agents.** We decompose performance by model and harness, analyze agent plans, patches, tests, and traversal behavior against the human development order, and quantify how depth and context compaction drive regressions, yielding actionable signals for harness design and model

Table 1: Comparison of HORIZONFLOW with existing coding benchmarks, grouped into the SWE family and the long horizon family. **Multi**: task spans multiple development units. **DAG**: explicit dependency DAG between units. **Pch**: average gold patch size in lines. **Tst**: average tests per task. **Time**: average human completion time per task.

Benchmark	Harness	Lng	Task Form	Multi	DAG	Pch	Tst	Time
HORIZONFLOW	Flow aware	8	dependency DAG	✓	✓	1,631	44	6.6m
SWE family								
SWE-bench	F2P only	1	issue	×	×	33	9	24.6d
Multi-SWE-bench	F2P only	7	issue	×	×	107	21	18.5d
SWE-bench Pro	F2P only	11	issue / feature	×	×	462	38	1.7m
FeatureBench	F2P only	1	feature	partial	×	1,256	38	19.5d
SWE-EVO	F2P only	1	software evolution	partial	×	610	31	7.1d
Long horizon family								
MLE-Bench	F2P only	1	repo generation	×	×	514	1	3.0m
NL2Repo-Bench	F2P only	1	repo generation	partial	×	1,623	138	3.4y
PaperBench	F2P only	1	repo generation	partial	×	8,316	20	100h
ProgramBench	F2P only	1	repo generation	partial	×	2,104	52	12.4d
RepoZero	F2P only	1	repo generation	partial	×	3,287	68	21.7d

training.

2 HORIZONFLOW: A Scalable Pipeline for Structured Long Horizon Coding Tasks

As illustrated in Figure 2, we propose HORIZONFLOW, a scalable automated pipeline that constructs structured long horizon coding tasks through five stages. **(1) Task Collection** harvests raw data from three authentic real world sources. **(2) Task Preprocessing** normalizes the raw data into atomic candidate tasks. **(3) Task Selection** filters candidates by two source agnostic thresholds. **(4) Task Instrumentation** attaches an executable environment, a formalized instruction, and a test suite to each surviving candidate. **(5) Intra Task Relation Construction** recovers the dependency DAG and attaches per unit requirement bundles.

2.1 Task Collection

We collect raw data from three authentic real world sources to span a broad spectrum of coding domains and a graded range of task difficulty, since a benchmark made entirely of the hardest tasks reveals less about agent behavior across the realistic difficulty surface than one that exercises the agent across both. **(1) Course Labs.** We crawl 112 university programming projects from the official course homepages of 30 universities and from public GitHub mirrors of their assignment releases, retrieving each as its complete native bundle of handout, skeleton code, and instructor side reference tests. **(2) PR Sequences.** We pull the full commit history and merged pull request stream of 56 actively maintained GitHub repositories, all under continuous community development with high star counts and long maintenance windows. **(3) Research Evolutions.** We gather 212 papers from 48 venues together with the arXiv tex of their cited prior work and forward citers, restricted to papers that are simultaneously highly cited and accompanied by a fully open source official implementation under an OSI approved license. A reference filter retains the 95 papers connected by load bearing inheritance edges. The three sources jointly anchor the benchmark in authentic developer, educator, and researcher artifacts rather than fabricated specifications. Per source coverage appears in Appendix B.

2.2 Task Preprocessing

We normalize the raw data into atomic candidate tasks before selection, since the three sources arrive heterogeneous in granularity, language, and file layout.

Course Labs. ❶ unify handouts written in mixed languages into a single English version; ❷ reorganize scattered skeleton files under one task root. Each project becomes one candidate task, yielding 112 candidates across the collected projects.

PR Sequences. ❶ clone each repository with full git history and snapshot the first commit as the base codebase; ❷ crawl its merged pull requests and extract per pull request diffs split by role (e.g., test, source, auxiliary); ❸ hand the repository to a Claude Code (Anthropic, 2025b) agent that, under a strict skill we author for this stage, performs repository level exploration and emits aggregated segments validated against a fixed acceptance standard. The procedure yields 148 candidate tasks across the 56 repositories.

Research Evolutions. ❶ select a high impact early seed paper in the collected venue set and collect the arXiv tex of the seed together with its cited prior work and forward citers; ❷ run a Claude Code agent that classifies each reference as a shallow

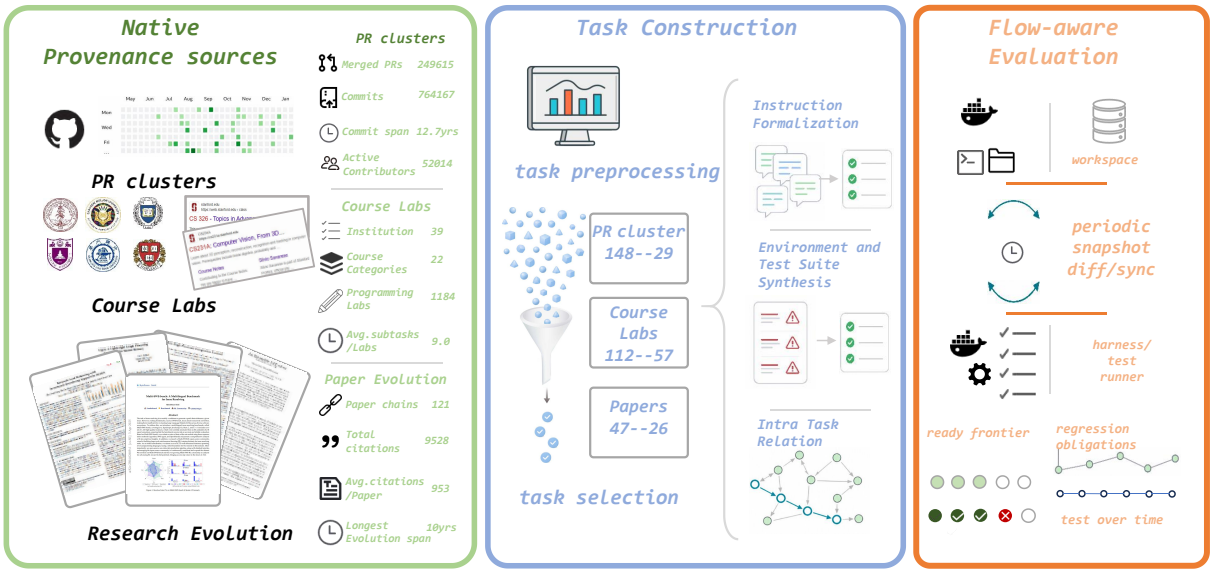

Figure 2: Overview of HORIZONFLOW. Three pipelines—PR Sequences, Course Labs, and Paper Lineages—yield tasks with explicit temporal dependencies, validated in a uniform sandbox.

related work mention or a load bearing inheritance edge; ❸ absorb every neighbor connected by a load bearing edge into the cluster and repeat the classification on the newly added papers until the closure admits no further additions. Each closed cluster becomes one candidate task, yielding 47 candidates across the 95 retained papers; 26 survive Selection.

Across all three sources we then wrap the heterogeneous native test runners behind a uniform task entry point, so that downstream stages observe a single invocation contract regardless of provenance. A worked end to end example of a single candidate task in HORIZONFLOW appears in Appendix D.

2.3 Task Selection

Candidate tasks pass two source agnostic thresholds. **Temporal Span** must reach at least 2.5 months, measured as the handout declared project duration for Course Labs, the merge time interval between the first and last commit of the region for PR Sequences, and the publication date span between the earliest and latest paper in the evolution for Research Evolutions. **Solution Scale** must reach at least 1,200, set per source because each source carries its own natural patch length and base codebase size, with per source values reported in Appendix B. After filtering, 112 tasks remain.

2.4 Task Instrumentation

We turn each selected candidate into a self contained long horizon evaluation instance with a formalized instruction that fixes the expected final state, a reproducible executable environment, and a discriminative test suite. The gold solution stays outside the agent's context and is exposed only through deterministic harness signals computed before any agent observation.

❶ **Instruction Formalization.** We treat the gold diff against the base codebase as ground truth and recover the developer's behavioral intent from it, since PR titles, course handouts, and paper method sections rarely carry enough signal on their own. A Claude Code agent (Anthropic, 2025b) backed by Claude Opus 4.7 (Anthropic, 2026) executes this recovery under a skill we author for this stage and produces a single outcome oriented instruction that fixes the expected final state without prescribing the path to reach it. The skill is bound by two objectives, (1) hide all evidence of the implementation path such as concrete edit locations and gold diff structure, and (2) expose only the externally visible contract that the test suite will bind to such as public signatures and required output formats. A symbol crosses this boundary into the instruction only when a test assertion cannot be written without naming it. The skill also strips repository names, course identifiers, paper titles, and author attributions so the source is unrecognizable on contact.

❷ **Environment and Test Suite Synthesis.** A Claude Code agent walks the source frame from upstream preprocessing, collects the dependencies the run requires, and writes a `Dockerfile` for the environment, with a companion `docker-compose.yaml` when the task spans multiple services. The agent then traverses the

DAG in topological order. At each unit u the agent drafts the missing tests on top of the current environment, applies u's gold patch on the recorded base, and runs the new tests under a three trial fail to pass check that admits $T(u)$ only when the full gold solution flips every test to pass (*solvability*), nothing flips on an empty workspace (*non triviality*), and the gold solution restricted to u's DAG ancestors still leaves at least one test failing (*discriminativeness* w.r.t. u's own contribution). A test that never executes is attributed to the environment and triggers a `Dockerfile` revision, and a test that runs but does not flip from fail to pass is attributed to the assertion and rewritten in place. After every unit has been visited, a closure pass replays the full gold patch against the accumulated suite on the recorded base. Units whose environment or suite cannot be repaired within the computational budget are discarded.

2.5 Intra Task Relation Construction

We formalize each task as a directed acyclic graph $G = (V, E)$ over its development units, with the four edge patterns we admit illustrated in Figure 5 of Appendix C. The four patterns we admit 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. An edge (u, v) is admitted only under unambiguous evidence on the gold artifacts. The first kind of evidence is hunk overlap on the same file. The second kind of evidence is symbol level dependence through an import, call, or subclass of an authoritative definition introduced by u. Spurious chains arising from hot monoliths, generated artifacts, and lockfiles are excluded by a denylist applied before edge admission, so that a file touched by most PRs in a repository contributes no edges and a regenerated lockfile never anchors a successor. The full admission rules appear in Appendix C. On 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 collects nodes whose tests all pass at t. Requirements compose along edges. The tests $T(v)$ are evaluated in a workspace in which every predecessor $T(u_i)$ remains enforced, so that satisfying v cannot regress a predecessor.

2.6 Evaluation Protocol

As illustrated in Figure 3, the agent receives the task instruction and per unit requirement bundles as plain files, with gold solutions and the identity of the ready unit hidden, and the harness records the visitation sequence σ for comparison against the human order σ^*.

Topologically Gated Test Release. We expose tests layer by layer along the dependency DAG, with every node of in degree greater than one acting as a gate. A gate node v joins the ready frontier only after all predecessors lie in C_t, and the tests of its descendants stay sealed until $T(v)$ flips from fail to pass. The gate governs only what the harness scores, so the agent is free to keep editing across layers and the gate never blocks its execution. Once v clears the gate, $T(v)$ together with the tests of every predecessor of v are kept enforced as regression tests on every later layer, so that subsequent edits are continuously probed for the agent's ability to maintain already finished units.

Dual Container Snapshot Pipeline. We separate edit execution from test adjudication across two containers because we run process oriented testing on top of an agent that keeps editing, and a single container would force the test process to share the agent's runtime and lose isolation from it. Container A hosts the agent's working tree and is the only container the agent can write to. Container B carries the full test suite and the reference environment, and reads from A only through snapshots pushed across by the harness. On a fixed time cadence the harness samples A and diffs the new snapshot against the queue head: an empty diff is dropped, while a non empty diff is appended to the queue and shipped into B for the currently exposed layer to run against. Snapshots therefore form a queue of monotonic edit states, and tests fire once per real change rather than once per sampling tick.

3 Experiments

To assess the capability of coding agents on long horizon software tasks, and to surface actionable signals for harness engineering and model tuning, we organize the experiments around three research questions. We evaluate on the full 112 task release of HORIZONFLOW.

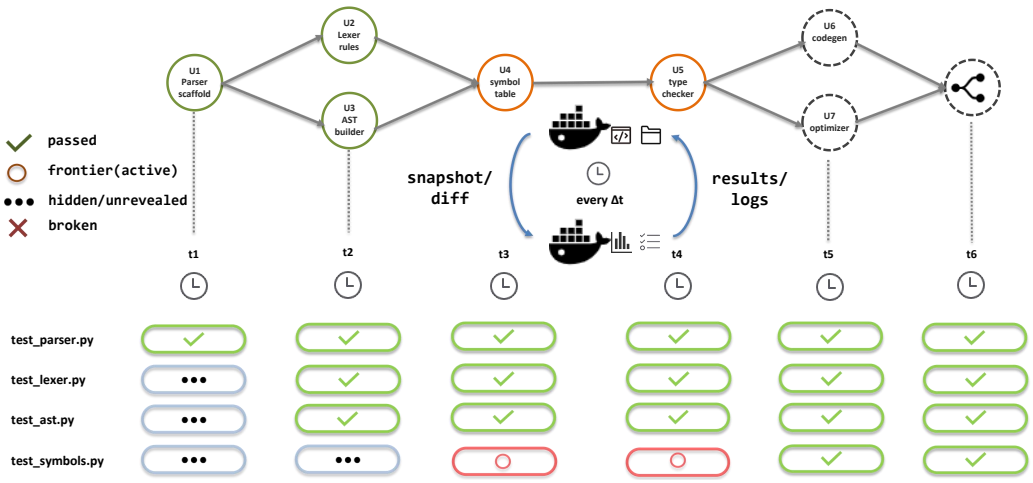

Figure 3: Schematic of the evaluation protocol.

3.1 RQ1: How Far Along Long-Horizon Coding Tasks Do Agents Sustain Progress?

Table 2 reports RQ1 results along three dimensions: generational scaling within each model family coupled with its own vendor scaffold, backbones swept under a fixed agent, and agents swept under a fixed backbone. Across all three, the highest cell still remains far from resolving HORIZONFLOW, and three points sharpen where the ceiling sits. **(i)** The strongest combination of model and scaffold we evaluate resolves only about a quarter of HORIZONFLOW, and absent the outer ralph loop a single invocation advances through merely a handful of dependency layers. Notably, this combination aligns each frontier model with the scaffold from the same vendor, under which the two are jointly designed and co adapted, yet both still warrant further adaptation toward long horizon tasks. **(ii)** The agent dimension exposes a limitation of current harness engineering. Absent an external loop, no scaffold sustains the model through long horizon execution by its own mechanisms. The agent occasionally resorts to selective avoidance, for instance emitting empty patches to feign closure or skipping prescribed order to cherry pick easier units. However, the external loop drives only nominal progress toward completion. Internal state maintenance within a scaffold and context continuity across invocations remain open design problems. **(iii)** In the model dimension, even under an outer ralph loop that compels every unit to be attempted, the Resolve Rate plateaus and late units fail at obligations carried from earlier failures. Agentic post training has lifted backbones from fragmentary text generation toward agentic execution, but the model still treats the task as a sequence of local issues rather than a global plan. We report pass@K in Appendix I to rule out single trial noise behind the within family monotonicity.

> **RQ1:** Long horizon coding is not solved at frontier scale; the ceiling is set by the agent's failure to treat the task as one global plan, not by per unit code generation.

3.2 RQ2: How Does Agent Behavior Differ from Human Development?

Table 3 compares agent trajectories against the human development order along the natural plan, solve, test progression, with the order recovered from source records as reference.

(1) Planning: an internal dependency gap. Agents cannot anticipate how a repository will evolve from the requirement set alone, and drop a large fraction of the pull request level dependencies that the human history actually obeys. Edge F1 and Layer ρ remain substantially lower than the human DAG for every harness, and degrade further from closed agents to open ones. Critical Path Ratio (CPR) and Width Ratio (WR) move in the opposite direction along the same axis. Industrial scaffolds invoke the model along a tree shaped concurrent structure and stay close to the human concurrency budget, whereas open agents driven by a linear loop collapse toward a near chain. The Width Ratio above unity for closed agents is not evidence of planning superior to the human one. Only a small share of those branches reflects a genuinely improved decomposition, while the rest scheduled in parallel what must be serial requirements that the agent missed. **(2) Solution: stylistic surplus over a comparable token base.** On units the agent does

Table 2: **RQ1 results. RR**: Resolve Rate. **TPR**: Test Pass Rate. **Depth**: normalized dependency layer depth. **Tok**: mean billed tokens per task in millions. Full definitions and settings in Appendix G and Appendix F.

Model	**RR↑**		**TPR↑**		**Depth↑**	**Tok↓**
	w/o	*w/*	*w/o*	*w/*		
GPT (Codex)						
GPT-5.5	14.29%	21.43%	39.62%	50.84%	0.53	8.36M
GPT-5.2	9.82%	13.39%	33.91%	43.90%	0.36	9.10M
GPT-5	5.36%	8.04%	28.45%	36.86%	0.22	8.92M
GPT-4o	1.79%	3.57%	21.85%	28.30%	0.09	4.51M
Claude (Claude Code)						
Opus-4.7	16.96%	25.00%	41.18%	53.05%	0.61	6.91M
Opus-4.5	12.50%	17.86%	36.97%	48.21%	0.46	5.38M
Sonnet-4.5	8.93%	12.50%	32.85%	42.58%	0.34	4.21M
Sonnet-4	4.46%	7.14%	27.43%	35.54%	0.18	3.76M
Qwen (Qwen Code)						
Qwen3.6-Plus	6.25%	9.82%	37.97%	45.10%	0.31	6.96M
Qwen3.5-Plus	5.36%	8.04%	31.75%	41.17%	0.19	6.88M
Qwen3-Max	0.89%	1.79%	17.45%	23.31%	0.07	5.71M
Qwen2.5-72B	0.00%	0.00%	10.61%	13.88%	0.03	5.02M

	RR↑		**TPR↑**		**Depth↑**	**Tok↓**
	w/o	*w/*	*w/o*	*w/*		
Model *(fixed agent: Claude Code)*						
Opus-4.7	16.96%	25.00%	41.18%	53.05%	0.61	6.91M
GPT-5.5	13.39%	20.54%	38.04%	49.30%	0.47	7.18M
GLM-5.1	13.39%	18.75%	34.55%	45.49%	0.44	4.37M
DeepSeek-V4P	11.61%	18.75%	36.07%	47.34%	0.46	3.59M
Gemini-3.1-Pro	8.93%	14.29%	32.86%	43.59%	0.31	4.99M
Qwen3.6-Plus	6.25%	9.82%	36.43%	47.50%	0.34	4.25M
Kimi-2.6	3.57%	6.25%	21.27%	28.33%	0.14	2.02M
Grok-4.1-FR	2.68%	4.46%	23.32%	30.97%	0.15	1.83M
Agent *(fixed model:* `gpt-5.4`*)*						
Codex	13.39%	18.75%	37.84%	49.06%	0.49	7.94M
Claude Code	12.50%	17.86%	36.97%	48.21%	0.42	7.03M
GitHub Copilot	10.71%	15.18%	34.21%	45.46%	0.38	8.26M
OpenHands	6.25%	9.82%	29.21%	39.30%	0.20	6.75M
SWE-agent	5.36%	8.93%	28.07%	38.11%	0.16	7.18M
mini-swe-agent	4.46%	7.14%	25.96%	35.20%	0.13	8.86M

Table 3: **Fused harness profile across RQ2 and RQ3. Edge F1**: prerequisite edge agreement. **Layer** ρ: Spearman correlation of topological layer indices. **CPR**: Critical Path Ratio. **WR**: Width Ratio. **PatchLen**: gold normalized patch length. **Jacc**: token Jaccard distance to gold. **#T**: tests authored. **F2P**: fail to pass yield. **Reg**: Regression Rate. **Rounds**: outer ralph loop rounds. Δ**TPR**$_r$: Test Pass Rate change across outer loop handoffs. **Ev/run**: compaction events per run. Δ**TPR**$_c$: Test Pass Rate change in the window around each compaction event. Full metric definitions, mechanism details, and counterfactual configurations in Appendix K, M, and N.

	RQ2: agent behavior vs human development									**RQ3: harness mechanisms**			
	Planning fidelity				**Solution**		**Testing**			**Ralph loop**		**Compaction**	
Agent	**Edge F1**	**Layer** ρ	**CPR**	**WR**	**PatchLen**	**Jacc**	**#T**	**F2P**	**Reg↓**	**Rounds**	Δ**TPR**$_r$	**Ev/run**	Δ**TPR**$_c$
Gold	1.00	1.00	0.40	1.00	1.00	0.00	44	1.00	0.00%	–	–	–	–
Closed-source agents													
Claude Code	0.71	0.65	0.31	1.08	1.58	0.62	28	0.47	7.11%	1.73	−0.42	4.47	−5.31
Codex	0.67	0.61	0.33	1.14	1.71	0.64	24	0.44	4.83%	1.58	−0.18	32.6	−3.74
GitHub Copilot	0.58	0.52	0.41	0.92	1.83	0.66	22	0.41	6.91%	2.08	0.00	–	–
Open-source agents													
OpenHands	0.39	0.33	0.85	0.39	2.31	0.73	16	0.34	2.46%	3.18	−2.07	3.21	−6.84
SWE-agent	0.37	0.31	0.88	0.36	2.36	0.74	15	0.33	2.18%	3.04	−1.76	2.94	−5.92
mini-swe-agent	0.27	0.22	0.97	0.24	2.54	0.76	11	0.28	0.24%	4.16	–	0.00	–

resolve, PatchLen exceeds gold by a sizable margin across all harnesses while Jacc to gold stays at a comparable moderate level, so patches are stylistically inflated rather than semantically off target. A diff level account of where the extra lines come from appears in Appendix K. **(3) Testing: a maintenance gap over the long horizon.** Agents do not maintain a dense test suite as a long horizon task unfolds, and the resulting sparsity leaves earlier obligations unguarded against later edits. Test count (#T) is markedly lower than that of the native suite for every harness, while fail to pass yield (F2P) on the tests that are produced is acceptable because each harness validates its candidate tests internally before committing them. Regression (Reg) is the downstream cost. Production agents drop several percent of previously satisfied obligations, and the apparently low Reg of lighter scaffolds is a denominator artifact because they accumulate too few passing obligations to regress against. Long horizon tasks inherently demand a dense test signal to monitor the running state of the harness and to provide signal for future training, and this is precisely what existing long horizon benchmarks do not offer.

RQ2: The long horizon gap is one of state discipline. Agents underspecify dependencies, overproduce code, and undertest, so what bounds them is the upkeep of their own plan, code, and test state.

3.3 RQ3: Which Harness Mechanisms Govern Long Horizon Task Performance?

Table 3 reports, for the two harness mechanisms that long horizon execution forces every scaffold to invoke, both the trigger frequency and the effect on task completion as measured by the disclosed test stream.

(1) The outer ralph loop handles the residual once a single invocation no longer suffices, with frequency in Rounds and completion effect in ΔTPR_r. Harnesses that rederive a fresh residual at each round transition incur few rounds and a small disclosed test change across the handoff, whereas harnesses that lean on an internalized goal mode accumulate more rounds and a more variable transition gap because revising a global plan mid run is more expensive than rederiving the residual. The dominant cost of long horizon execution therefore sits in residual drift across handoffs rather than in the rounds themselves. Residual rederivation is the safer default, and goal mode is best invoked selectively on subgraphs whose ready frontier is genuinely wide. **(2) The in context compaction mechanism** preserves earlier turns once the transcript saturates the window, with frequency in Ev/run and completion effect in ΔTPR_c. Negative ΔTPR_c tracks the mechanism rather than the event rate. Harnesses that summarize the prefix drop prerequisite invariants and lose disclosed tests in the window immediately following each event, harnesses that prune or truncate retain more of the surviving state at the cost of recovery tokens when context is reread, and the external file scratchpad never compacts the live transcript at all. The cluster of regressions around compaction events is the same long horizon mechanism behind the Reg column of the testing group, and a within harness counterfactual on aggressive versus delayed triggers rules out incidental alignment with hard task phases (Appendix N). The workable design is to externalize the surviving state into harness owned storage rather than entrust it to a model authored summary. The residual and the surviving state are the same object viewed at two horizons, and a long horizon harness should own both explicitly. Between rounds it should rederive the residual deterministically, and across compactions it should preserve a structured store of open obligations and last known passing tests, with the model authoring neither.

RQ3: Current harnesses already keep the agent running across the long horizon, but two mechanisms remain weak: task routing across outer loop handoffs and state retention across in context compaction.

4 Related Work

Function-level code-generation benchmarks (Chen et al., 2021) pioneered programmatic evaluation but score isolated snippets. Repository-level benchmarks centered on SWE-bench (Jimenez et al., 2023a) package each task as a single issue judged by a terminal verdict. Feature-level and evolution-level benchmarks (Zhou et al., 2026) enlarge task scope without organizing it around prerequisite structure, and broader process benchmarks (Kwa et al., 2025) extend past single patches yet do not retain completed work as live obligations. Modern coding agents and reasoning frameworks (Anthropic, 2025b; Yao et al., 2023) rely on persistent memory, planning, and context summarization to sustain extended sessions, yet harnesses surface only a final pass rate and hide trajectory-level signal. Software-engineering work on change impact and release planning (Lehnert, 2011) establishes dependency structure as a first-class artifact but does not connect it to executable evaluation. The full taxonomy appears in Appendix A.

5 Conclusion

HORIZONFLOW targets a capability that existing coding benchmarks evaluate only tangentially: whether an agent sustains correct forward progress once development is dependency-structured rather than issue-local. By formalizing tasks as DAGs, decoupling tests along graph edges, and converting completed-node tests into persistent regression obligations, the benchmark reorients evaluation from isolated patch success toward sustained engineering behavior. Empirically, performance degrades with dependency depth, regressions concentrate after context summarization, and agent trajectories diverge markedly from human graph traversal. Progress will therefore require not only stronger backbones but also better mechanisms for dependency-aware state tracking and regression-preserving context management. Scope conditions, source assumptions, and version pinning caveats are stated in the Limitations section.

Limitations

The recovered DAG is a lower bound on prerequisite structure: we admit edges only under four source evidenced patterns and a hot file denylist, so depth and width statistics are conservative. The three pipelines admit tasks only when commit history, curriculum decomposition, or citation closure yields verifiable evidence, so mobile, frontend heavy, and hardware near projects sit outside the current pool. Correctness is read off the committed test suite, so TPR, Resolve Rate, and Regression Rate should be read against that instrument.

References

Alibaba Cloud. 2025. Alibaba cloud unveils strategic roadmaps for the next generation of 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. Includes Qwen3-Max. Accessed 2026-05-16.

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

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

Anthropic. 2025b. Claude code. Anthropic product page.

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

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

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

Anysphere. 2024. Cursor: The AI-first code editor. https://www.cursor.com.

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, and 1 others. 2024. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pages 17682–17690.

Nicholas Carlini. 2026. Building a C compiler with a team of parallel claudes. Anthropic Engineering blog.

Guoxin Chen, Fanzhe Meng, Jiale Zhao, Minghao Li, Daixuan Cheng, Huatong Song, Jie Chen, Yuzhi Lin, Hui Chen, Xin Zhao, and 1 others. 2026. Beyondswe: Can current code agent survive beyond single-repo bug fixing? *arXiv preprint arXiv:2603.03194*.

Haorui Chen, Chengze Li, and Jia Li. 2025. Featbench: Evaluating coding agents on feature implementation for vibe coding. *arXiv preprint arXiv:2509.22237*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Codeium. 2025. Windsurf: The AI-powered IDE. https://windsurf.com.

Cognition. 2026. Introducing devin. Devin Docs.

Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uirá Kuber, Yasutaka Kamei, and Ahmed E. Hassan. 2016. A framework for evaluating the results of the SZZ approach for identifying bug-introducing changes. In *IEEE Transactions on Software Engineering*, volume 43, pages 641–657. IEEE.

DeepSeek-AI. 2026. DeepSeek-V4-Pro model card. https://huggingface.co/deepseek-ai/DeepSeek-V4-Pro. Accessed 2026-05-16.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, and 1 others. 2025. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*.

Jingzhe Ding, Shengda Long, Changxin Pu, Huan Zhou, Hongwan Gao, and 1 others. 2025. Nl2repo-bench: Towards long-horizon repository generation evaluation of coding agents. *arXiv preprint arXiv:2512.12730*.

Nicholas Edwards, Yukyung Lee, Yujun Audrey Mao, Yulu Qin, Sebastian Schuster, and Najoung Kim. 2025. RExBench: Can coding agents autonomously implement AI research extensions? *arXiv preprint arXiv:2506.22598*.

Yukang Feng, Jianwen Sun, Zelai Yang, Jiaxin Ai, Chuanhao Li, and 1 others. 2026. Longcli-bench: A preliminary benchmark and study for long-horizon agentic programming in command-line interfaces. *arXiv preprint arXiv:2602.14337*.

Dhruv Gautam, Spandan Garg, Jinu Jang, Neel Sundaresan, and Roshanak Zilouchian Moghaddam. 2025. Refactorbench: Evaluating stateful reasoning in language agents through code. *arXiv preprint arXiv:2503.07832*.

GitHub. 2026a. Github copilot cloud agent. GitHub Docs.

GitHub. 2026b. Github copilot: Your ai pair programmer. https://github.com/features/copilot. Accessed: 2026-04-26.

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

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and 1 others. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, and 1 others. 2023. Metagpt: Meta programming for a multi-agent collaborative framework. In *The twelfth international conference on learning representations*.

Geoffrey Huntley. 2025. Ralph wiggum as a "software engineer". Accessed: 2026-05-07.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023a. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2023b. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

Thomas Kwa, Ben West, Joel Becker, Amy Deng, Katharyn Garcia, Max Hasin, Sami Jawhar, Megan Kinniment, Nate Rush, Sydney Von Arx, and 1 others. 2025. Measuring ai ability to complete long tasks. *arXiv preprint arXiv:2503.14499*, 352.

Steffen Lehnert. 2011. A taxonomy for software change impact analysis. *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution*, pages 41–50.

Bowen Li, Wenhan Wu, Ziwei Tang, Lin Shi, John Yang, Jinyang Li, Shunyu Yao, Chen Yin, Yixin Liu, Hao Peng, and 1 others. 2024. DevBench: A comprehensive benchmark for software development. *arXiv preprint arXiv:2403.08604*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in neural information processing systems*, 36:21558–21572.

Yuxiang Liu and 1 others. 2025. SWE-bench-M: A benchmark for mobile software engineering agents. *arXiv preprint*.

Mike A Merrill, Alexander G Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E Kelly Buchanan, and 1 others. 2026. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*.

Samuel Miserendino, Michele Wang, Tejal Patwardhan, and Johannes Heidecke. 2025. 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*, pages 44412–44450.

Moonshot AI. 2026. Kimi K2.6: From code to creation, from one to many. https://www.kimi.com/ai-models/kimi-k2-6. Use for the RQ1 table entry named Kimi-2.6 unless the intended model is changed. Accessed 2026-05-16.

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

OpenAI. 2024b. Introducing swe-bench verified. OpenAI blog.

OpenAI. 2025a. Introducing codex. OpenAI product announcement.

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

OpenAI. 2025c. 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. Accessed 2026-05-16.

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

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

OpenAI. 2026c. Why swe-bench verified no longer measures frontier coding capabilities. OpenAI research note.

Gabriel Orlanski, Devjeet Roy, Alexander Yun, Changho Shin, Alex Gu, Albert Ge, Dyah Adila,

Frederic Sala, and Aws Albarghouthi. 2026. Slopcodebench: Benchmarking how coding agents degrade over long-horizon iterative tasks. *arXiv preprint arXiv:2603.24755*.

Charles Packer, Vivian Fang, Shishir_G Patil, Kevin Lin, Sarah Wooders, and Joseph_E Gonzalez. 2023. Memgpt: towards llms as operating systems.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. 2024. Training software engineering agents and verifiers with swe-gym. *arXiv preprint arXiv:2412.21139*.

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

Qwen Team. 2026. Qwen3.6-Plus: Towards real world agents. https://qwen.ai/blog?id=qwen3.6. Accessed 2026-05-16.

Günther Ruhe. 2005. *The Art and Science of Software Release Planning*. Springer.

Atharva Sehgal, James Hou, Swarat Chaudhuri, Jennifer J Sun, and Yisong Yue. Formulacode: Evaluating agentic superoptimization on large codebases. In *ICML 2025 Workshop on Programmatic Representations for Agent Learning*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652.

Zachary S Siegel, Sayash Kapoor, Nitya Nagdir, Benedikt Stroebl, and Arvind Narayanan. 2024. Core-bench: Fostering the credibility of published research through a computational reproducibility agent benchmark. *arXiv preprint arXiv: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, and 1 others. 2026. Omnicode: A benchmark for evaluating software engineering agents. *arXiv preprint arXiv:2602.02262*.

Alvin W Tan, Sunny Yu, Bria Long, Wanjing Anya, Tonya Murray, Rebecca D Silverman, Jason D Yeatman, and Michael C Frank. 2024. Devbench: A multimodal developmental benchmark for language learning. *Advances in Neural Information Processing Systems*, 37:77445–77467.

Minh VT Thai, Tue Le, Dung Nguyen Manh, Huy Phan Nhat, and Nghi DQ Bui. 2025. Swe-evo: Benchmarking coding agents in long-horizon software evolution scenarios. *arXiv preprint arXiv:2512.18470*.

Qiang Tu and Michael W. Godfrey. 2002. The evolution of large C programs. In *Science of Computer Programming*, volume 45, pages 203–225. Elsevier.

Harold Valdivia Garcia and Emad Shihab. 2018. Characterizing and predicting blocking bugs in open source projects. *Journal of Systems and Software*, 143:44–58.

Sanidhya Vijayvargiya, Xuhui Zhou, Akhila Yerukola, Maarten Sap, and Graham Neubig. 2025. Ambigswe: Interactive agents to overcome underspecificity in software engineering. *arXiv preprint arXiv:2502.13069*.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, and 1 others. 2024. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*.

Hjalmar Wijk, Tao Lin, Joel Becker, Sami Jawhar, and 1 others. 2025. Re-bench: Evaluating frontier ai r&d capabilities of language model agents against human experts. In *ICML 2025 Workshop / OpenReview version*.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, and 1 others. 2024. Autogen: Enabling next-gen llm applications via multi-agent conversations. In *First conference on language modeling*.

xAI. 2026. Grok 4.1 Fast Reasoning. https://docs.x.ai/developers/models/grok-4-1-fast-reasoning. Accessed 2026-05-16.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024a. SWE-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R. Narasimhan, and Ofir Press. 2024b. Swe-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*.

John Yang, Carlos E Jimenez, Alex L Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R Narasimhan, and 1 others. 2024c. Swe-bench multimodal: Do ai systems generalize to visual software domains? *arXiv preprint arXiv:2410.03859*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822.

Z.AI. 2026. GLM-5.1 overview. https://docs.z.ai/guides/llm/glm-5.1. Accessed 2026-05-16.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, and 1 others. 2025. Multi-swe-bench: A multilingual benchmark for issue resolving. *arXiv preprint arXiv:2504.02605*.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. *arXiv preprint arXiv:2404.05427*.

Qixing Zhou, Jiacheng Zhang, Haiyang Wang, Rui Hao, Jiahe Wang, Minghao Han, Yuxue Yang, Shuzhe Wu, Feiyang Pan, Lue Fan, and 1 others. 2026. Featurebench: Benchmarking agentic coding for complex feature development. *arXiv preprint arXiv:2602.10975*.

Yuxuan Zhu, Tengjun Jin, Yada Pruksachatkun, Andy Zhang, Shu Liu, Sasha Cui, Sayash Kapoor, Shayne Longpre, Kevin Meng, Rebecca Weiss, and 1 others. 2025. Establishing best practices for building rigorous agentic benchmarks. *arXiv preprint arXiv:2507.02825*.

A Related Coding Benchmarks, Agents, and Reasoning Frameworks

Table 4 groups the full set of prior work surveyed in Section 4 by topic, with one representative reference per row and a one-line note on how the cited work differs from the long-horizon, DAG-structured contract introduced by HORIZONFLOW. The body paragraph in Section 4 cites only the lead exemplar of each cluster; this appendix carries the rest so that the reader can locate each individual prior work without inflating the main-text citation density.

Table 4: Prior work clustered by topic. Each cluster's lead reference (left description) is the exemplar cited inline in Section 4; the right column carries the full enumeration including extensions, follow-ups, and parallel proposals.

Cluster	References (representative + extensions)
Function-level code-gen. Score isolated snippets against unit tests.	Chen et al. (2021); Hendrycks et al. (2021); Jain et al. (2024); Liu et al. (2023)
SWE-bench family. Repository-level, single-issue, terminal verdict.	Jimenez et al. (2023a,b); Zan et al. (2025); Yang et al. (2024c); Liu et al. (2025); Deng et al. (2025); OpenAI (2024b); Miserendino et al. (2025); Pan et al. (2024); Wijk et al. (2025); Sonwane et al. (2026); Sehgal et al.; Tan et al. (2024)
SWE-bench audits. Document contamination and overfitting in verified variants.	OpenAI (2026c); Zhu et al. (2025)
Feature / evolution. Enlarge task scope but lack explicit prerequisite structure.	Zhou et al. (2026); Chen et al. (2025); Thai et al. (2025); Gautam et al. (2025); Orlanski et al. (2026); Feng et al. (2026); Ding et al. (2025)
Process / trajectory. Extend evaluation past a single patch but do not keep completed work as live obligations.	Kwa et al. (2025); Siegel et al. (2024); Li et al. (2024); Merrill et al. (2026); Edwards et al. (2025); Chen et al. (2026); Vijayvargiya et al. (2025)
Coding agents. Wrappers around frontier LLMs we evaluate or contrast with.	Anthropic (2025b); OpenAI (2025a); Cognition (2026); Wang et al. (2024); Yang et al. (2024a,b); Zhang et al. (2024); Xia et al. (2024); GitHub (2026a); Huntley (2025)
Reasoning frameworks. Provide planning / memory / context-summarization primitives that the agents above build on.	Yao et al. (2023); Besta et al. (2024); Shinn et al. (2023); Hong et al. (2023); Wu et al. (2024); Wang et al. (2023); Packer et al. (2023)
SE prerequisites. Dependency structure, PR ordering, release planning treated as first-class but not connected to executable evaluation.	Lehnert (2011); Tu and Godfrey (2002); Costa et al. (2016); Valdivia Garcia and Shihab (2018); Ruhe (2005)

B Per-Task Statistics

The 112 HORIZONFLOW 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), spanning twenty six leaf subcategories in total.

Tables 5 and 6 report per-task structural statistics for all 112 tasks in HORIZONFLOW: repository scale (#Files, #LoC), unit count (#Num development units), task-structure summaries (average #Modules, average DAG edges, average dependency depth), and gold-implementation targets (average #Lines, #Files, #LoC). The split into two tables is purely a typesetting convenience; the row sets are disjoint and union to the full 112-task pool.

Table 5: Statistics of the HORIZONFLOW task structure and implementation targets (Part 1 of 2; the remaining 56 tasks are in Table 6). Columns: **Task** task slug; **#Files**, **#LoC** repository scale (files post-patch, gold-patch diff lines); **#Num** development units; **#Mod**, **#Edges**, **Depth** task-DAG structural summaries (modules, dependency edges, longest chain); **Avg. LoC** mean diff lines per unit; **Max LoC** largest single-unit patch; **Width** maximum concurrent units at any DAG layer (parallelism degree).

Repository			Units	Task structure			Implementation target		
Task	**#Files**	**#LoC**	**#Num**	**#Mod**	**#Edges**	**Depth**	**Avg. LoC**	**Max LoC**	**Width**
task_ClickHouse_seg07	3.3k	505k	825	30	2123	34	612	71.4k	410
task_NodeBB_seg01	1.7k	396k	134	17	640	27	3.0k	64.2k	25
task_NodeBB_seg03	4.1k	294k	71	22	130	10	4.1k	35.5k	31
task_NodeBB_seg05	4.6k	130k	355	20	127	10	365	11.3k	310
task_TypeScript_seg01	14.4k	379k	406	18	1119	47	934	95.6k	304
task_afl_fuzzing	11	5142	5	5	4	2	1.0k	1400	4
task_alloy_formal_modeling	18	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	5.0k	50.3k	303	20	727	17	166	11.8k	130
task_ansible_seg13	5.5k	26.9k	94	16	59	5	286	6010	58
task_arch_labs_adder_riscv_cache	139	614	10	10	7	3	61.4	192	6
task_architecture_riscv_labs	420	1130	15	15	7	3	75.3	273	9
task_boogie_dafny	21	7030	3	3	2	3	2.3k	2811	1
task_calcite_query_optimization	9	6367	3	3	2	2	2.1k	2741	2
task_cocos2dx_navmesh_medium	4.5k	5207	14	14	12	4	372	1143	9
task_cocos2dx_physics2d_medium	4.2k	4700	18	18	21	7	261	503	9
task_compiler_cmm_mips	29	2688	9	9	7	4	299	1839	6
task_compiler_fdmj_llvm	89	1304	7	7	5	4	186	472	2
task_compiler_rustlike_rv32	97	2079	6	6	6	4	346	606	2
task_compiler_sysy_rust	21	1906	6	6	5	4	318	591	2
task_cpachecker	21	5405	3	3	2	3	1.8k	2243	1
task_cpp_warmup_graphics	658	388	4	4	2	2	97.0	123	2
task_cs61b_extra_java_bundle	218	2837	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	1977	15	15	4	2	132	330	11
task_db_nanodb	417	1522	9	9	8	3	169	774	4
task_db_storage_index_labs	240	2398	10	10	11	7	240	1374	3
task_dcqcn_rdma_evolution	2.3k	353k	11	11	17	7	32.1k	353k	3
task_deep_learning_three_assignments	84	980	10	10	5	3	98.0	202	6
task_deep_rl_coursework	203	3285	19	19	23	6	173	811	6
task_dist_sys_raft_go	89	1173	4	4	5	3	293	581	2
task_distributed_fs_chfs	2.1k	2875	12	12	15	7	240	1195	3
task_django_seg01	5.8k	16.5k	209	16	476	21	78.8	1687	45
task_django_seg05	6.6k	15.9k	221	16	357	15	71.8	1690	108
task_django_seg11	7.0k	11.7k	171	17	224	12	68.4	1121	82
task_doop_souffle_static_analysis	17	6342	5	5	4	3	1.3k	1458	2
task_ds_five_assignments	27	4914	5	5	0	1	983	1231	5
task_echarts_seg01	1.3k	20.2k	50	18	140	14	404	3903	15
task_echarts_seg04	1.4k	26.3k	53	20	153	13	496	12.6k	22
task_echarts_seg05	1.5k	14.2k	43	20	140	12	330	3271	20
task_echarts_seg07	1.5k	10.7k	67	10	229	13	160	2102	31
task_echarts_seg08	1.6k	28.3k	77	11	340	20	367	4742	32
task_echarts_seg09	1.6k	6779	46	11	128	12	147	1037	26
task_echarts_seg10	1.6k	17.6k	48	24	80	9	367	2229	31
task_echarts_seg11	1.6k	12.4k	62	38	215	17	200	1679	31
task_echarts_seg13	1.9k	8098	48	34	66	7	169	1509	27
task_evosuite_test_generation	17	8216	6	6	6	3	1.4k	1434	3
task_frama_c	21	4629	3	3	2	3	1.5k	2065	1
task_framework_seg04	1.4k	109k	391	18	787	22	278	15.0k	172
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	5239	11	11	12	5	476	1261	7
task_graph_processing	6	5371	3	3	2	3	1.8k	1819	1
task_graphics_scotty3d	390	1199	8	8	2	3	150	276	6
task_hadoop_seg05	36.3k	526k	318	16	597	16	1.7k	103k	142
task_intro_ai_pacman	431	662	11	11	7	3	60.2	98	5

Table 6: Statistics of the HORIZONFLOW task structure and implementation targets (Part 2 of 2; continued from Table 5). Column definitions are identical to Part 1.

Repository			Units	Task structure			Implementation target		
Task	**#Files**	**#LoC**	**#Num**	**#Mod**	**#Edges**	**Depth**	**Avg. LoC**	**Max LoC**	**Width**
task_java_ds_coursework	96	3997	17	17	10	4	235	522	6
task_jenkins_seg02	11.4k	494k	84	11	167	16	5.9k	43.5k	40
task_jenkins_seg05	12.3k	63.6k	46	7	48	7	1.4k	13.1k	23
task_jpf_symbolic_execution	1.2k	229k	3	3	2	3	76.4k	126k	1
task_ligra_graph_evolution	56	14.7k	8	8	8	4	1.8k	14.7k	3
task_linux011_kernel_hacks	137	2690	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	1841	6	6	6	4	307	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	575	6	6	3	2	95.8	233	4
task_mlsys_autodiff_pa1	9	639	5	5	5	4	128	215	2
task_monogame_math2d_medium	2.1k	10.0k	12	12	17	4	834	5002	4
task_monogame_transform3d_medium	2.1k	4481	8	8	12	4	560	2318	2
task_mysql_sql_labs	48	1093	9	9	7	3	121	413	5
task_nandteris	256	865	6	6	8	5	144	276	2
task_navidrome_seg02	366	52.1k	23	11	9	3	2.3k	11.0k	16
task_navidrome_seg04	572	29.1k	38	12	16	5	766	13.1k	27
task_navidrome_seg05	628	50.5k	36	12	14	4	1.4k	17.4k	23
task_navidrome_seg07	753	7595	15	12	4	3	506	1448	12
task_net_os_labs	54	2019	18	18	10	6	112	146	9
task_networks_proxy_rdt	48	873	7	7	2	2	125	299	5
task_node_seg03	37.7k	126k	320	24	358	15	394	49.6k	209
task_ns3_network_simulation	3.2k	337k	2	2	1	2	168.3k	190k	1
task_numpy_dl_aics_style	176	900	6	6	6	4	150	272	2
task_opengl_glfw_labs	1.4k	1259	8	8	7	8	157	274	1
task_os_c_fs_labs	28	1230	7	7	1	2	176	639	6
task_os_distributed_practicals	19	1049	5	5	0	1	210	595	5
task_os_labs_asm_kernel	780	14.3k	11	11	9	5	1.3k	3874	4
task_panda3d_collision_hard	6.6k	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	6139	5	5	5	3	1.2k	1332	3
task_rails_seg09	4.8k	11.4k	122	16	90	7	93.7	1146	78
task_raylib_shapes2d_medium	1.4k	2926	5	5	5	3	585	2647	3
task_raylib_textures_hard	1.4k	4891	9	9	10	3	543	1241	4
task_rdma_ns3_evolution	2.3k	357k	12	12	23	9	29.8k	357k	3
task_riscv_cpu_six_labs	305	3429	6	6	6	3	572	720	4
task_riscv_cpu_sv	133	1875	12	12	17	6	156	449	5
task_riscv_cpu_verilog_three	56	1157	6	6	6	5	193	275	2
task_riscv_os_multiproject	619	4254	7	7	5	6	608	2006	2
task_sat_minisat_glucose	29	17.3k	9	9	11	8	1.9k	17.3k	2
task_security_labs	79	727	7	7	5	4	104	308	4
task_sicp_py_scheme_sql	696	3135	22	22	25	13	142	698	5
task_sklearn_ml_toolchain	4	5232	3	3	3	3	1.7k	2445	1
task_soot_taint_analysis	10	5187	3	3	2	3	1.7k	1919	1
task_spring_course_mgmt_fullstack	19.8k	730	7	7	8	3	104	242	3
task_sql_engine_myjql	42	928	3	3	1	2	309	524	2
task_stream_mining	5	5245	5	5	5	3	1.0k	1241	2
task_systems_csapp_labs	101	1289	6	6	1	2	215	323	5
task_systems_four_labs	342	1913	10	10	5	3	191	623	6
task_tcp_course_stack	48	1405	2	2	1	2	702	1192	1
task_vim_editor_ds	40	462	5	5	3	3	92.4	308	3
task_x86_os_four_labs	128	2229	9	9	11	6	248	600	2
task_xjqkl	26	439	5	5	4	3	87.8	152	3
task_z3_fixedpoint_verification	11	6074	5	5	5	4	1.2k	1345	2

C Task Schema and Harness Invocation

```
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
```

Each task ships as a self-contained directory with the following layout: `Dockerfile` and `docker-compose.yaml` pin a reproducible execution environment; `run-tests.sh` is the standard test entry point invoked by the harness at every checkpoint; `solution.sh`/`gold-patch.diff` hold the reference human solution used to validate the test suite and to anchor solution-quality comparisons; `task.yaml` declares per-unit metadata (requirement text, file/symbol scope, dependency parents, attached tests); and the `tests/` subtree holds the source-derived and supplemented behavioral tests, indexed back to their adjudicating unit. The harness invocation contract is uniform across the three task sources: the harness mounts the task directory into the container, exposes *all* per-unit requirement manifests to the agent up front (no requirement is ever hidden), runs `run-tests.sh` after every agent commit, and records the resulting test-state vector $s_i^{(\tau)}$ used by every metric in Appendix G. The *ready frontier* concept governs only which tests are scored as active obligations at each checkpoint, not what the agent can read.

D A Real Task Example

To make the schema described in Section 2 and Appendix C concrete, we walk through one materialized task end to end. We select a PR Sequences task drawn from an actively maintained open source repository, since this source exercises every stage of the pipeline (commit history snapshotting, per pull request diff extraction by role, repository level skill driven exploration, dependency DAG recovery). Identifiable tokens have been redacted in line with the contamination protocol of Appendix E; the redacted fields are marked `<repo>`, `<unit-id>`, and `<author>`.

On-disk layout of a single HORIZONFLOW task

`<repo>-<task-id>/`	*self-contained task bundle*
`Dockerfile`	*pinned base image, system deps, lockfile*
`docker-compose.yaml`	*multi-service tasks only*
`run-tests.sh`	*harness entry point at every checkpoint*
`solution.sh`	*driver that applies the reference solution*
`gold-patch.diff`	*reference human solution as a unified diff*
`task.yaml`	*global task metadata and DAG topology*
`units/`	*per-unit manifests, all visible to the agent*
`unit_001.yaml`	*requirement, scope, parents, attached tests*
`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 4: **Schema of a materialized HORIZONFLOW task.** The bundle is uniform across PR Sequences, Course Labs, and Research Evolutions. `task.yaml` carries the dependency DAG over units; `units/unit_*.yaml` carries per-unit requirements and test bindings; `tests/unit_*/` groups tests by their adjudicating unit so the harness can resolve the active-obligation set from the ready frontier at each checkpoint.

Native artifact. The task originates from a sequence of **9** merged pull requests against `<repo>` over a temporal span of **4.2 months**, touching **37** source files and **14** test files. The base codebase at the entry commit comprises **18.4k** LoC across **142** files. Each pull request arrives with a description, a per file diff, and a CI run record from the upstream test suite.

Preprocessing output. After Section 2.2, the nine pull requests decompose into **12** atomic can-

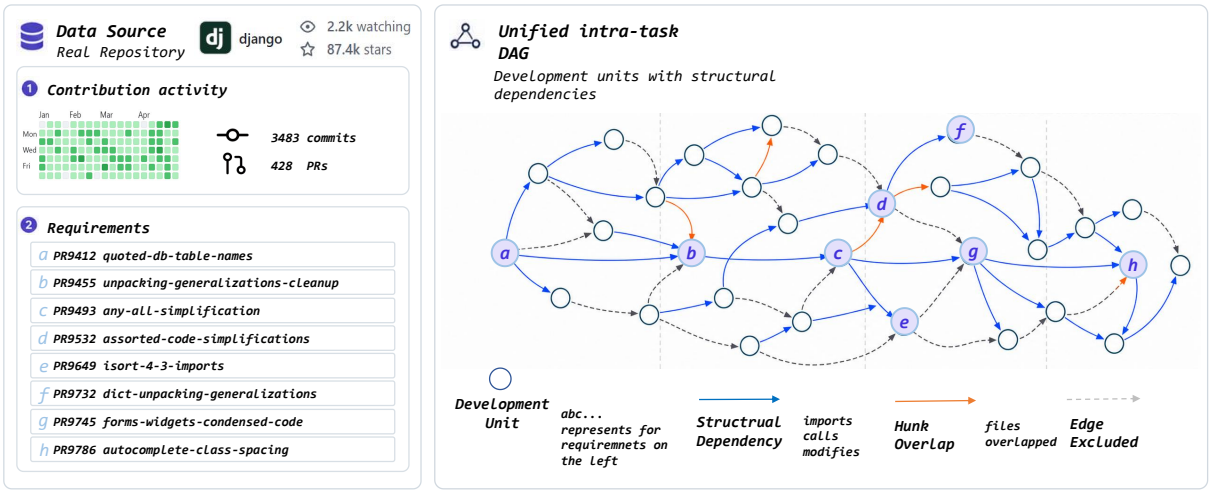

Figure 5: Four temporal-dependency patterns illustrated on one Ansible task's construction process: sequential PR chains, structural module reuse, functional producer–consumer API edges, and compositional layering. Edges come from the task's native provenance.

didate units. Three of the original PRs are split because their diffs touch disjoint feature surfaces; two adjacent PRs are merged because the second only patches a regression introduced by the first and carries no independent behavioral contract. The Claude Code agent emits the aggregated segments under our strict skill and they pass the fixed acceptance standard on the first attempt.

Materialized task bundle. The candidate is rewritten into the uniform schema of Appendix C. The resulting on disk layout is:

Task bundle layout

```
<repo>-<unit-id>/
  Dockerfile
  docker-compose.yaml
  run-tests.sh
  solution.sh
  gold-patch.diff
  task.yaml
  tests/
    unit_001/...
    unit_002/...
    ...
    unit_012/...
```

The `Dockerfile` pins Python 3.11.6, three system packages absent from the upstream README but recovered by the agent from CI configuration, and a lockfile derived from the entry commit. The harness reports exit code 0 on `gold-patch.diff` after a single build attempt.

Per unit requirement. A representative unit's `task.yaml` entry (paraphrased and redacted) reads:

Per unit requirement (excerpt)

```
unit_id: <unit-id>
requirement: |
  Extend the streaming response handler so
that
  partial chunks emitted before a terminating
  sentinel are buffered into a single payload
  visible to downstream consumers, preserving
  ordering across concurrent producers.
file_scope:
  - src/stream/handler.py
  - src/stream/buffer.py
symbol_scope:
  - StreamHandler.consume
  - StreamHandler._flush
acceptance:
  - concurrent producers preserve emission
order
  - sentinel terminates buffering atomically
  - downstream observers see one payload per
    logical message
parents: [unit_003, unit_005]
tests:
  - tests/unit_007/test_ordering.py
  - tests/unit_007/test_sentinel.py
  - tests/unit_007/test_concurrent.py
```

The requirement is phrased as a behavioral contract and acceptance conditions only. We explicitly avoid procedural phrasing, command level instructions, and references to the gold patch.

Recovered DAG. Intra task relation construction (Section 2.5) admits **18** edges over the 12 units under the unambiguous evidence rules of Appendix C. The longest dependency chain has depth **6**; the maximum out degree is **4** (a foundational refactor unit that later units import from). Three units form an independent branch with no incoming edges from the rest of the task, reflecting a parallel feature surface introduced midway through the PR sequence.

Trial verification. The materialized test suite passes the solvability, non-triviality, and discriminativeness trials of Section 2.4 on the first run for **10** of **12** units. Two units require one round of agent driven repair: in one case a flaky timing assertion is replaced with a deterministic state check; in the other a discriminativeness trial reveals that two predecessor tests already cover the unit's full surface, and the agent strengthens the unit suite with an additional assertion targeting the new behavior.

Evaluation trace excerpt. Under the protocol of Section 2.6, an agent's trajectory on this task records **31** checkpoints over the development episode. The visitation sequence σ differs from the human order σ^* on **3** of 12 units, all within branches whose topological order is not totally determined by the DAG. Two regressions are recorded against previously completed units and one is recovered by the agent before episode end. The full trajectory matrix S is omitted here for space; the metrics derived from S appear in Appendix G.

E On Data Contamination and Memorization

Long-horizon coding benchmarks built from public sources (PR Sequences, Course Labs, Research Evolutions) inevitably overlap with the pre-training data of modern LLMs. We address this risk with a two-layer protocol.

Input-side anonymization (prompt-level defense). During Task Instrumentation (Section 2.4), every unit requirement is manually rewritten to remove identifiable tokens that would let a model recognize the source on contact: repository names, course identifiers (e.g., MIT 6.824, Stanford CS144), paper titles, and author attributions. The same rewriting policy is applied at the segment (SEG) level for PR Sequences, where the original commit messages and PR titles, which often name the feature, the issue tracker ID, or the merging author, are stripped and replaced with the rewritten requirement. After this step, the prompt presented to an agent contains only the behavioral specification, function signatures, and tests; the surface evidence that would allow a model to retrieve a memorized solution by name is removed.

Capability-ceiling argument (weight-level memorization is not the bottleneck). Anonymization addresses prompt-level identification but cannot remove training-set memorization from model weights: an LLM that has internalized, say, a reference Raft implementation will still produce the algorithm regardless of how the prompt is phrased. We argue that for HORIZONFLOW this residual risk does not materially threaten the benchmark's validity, on the following grounds.

The strongest configuration in our evaluation, Claude Code with Opus-4.7, an Anthropic-native pairing on a frontier backbone with the most permissive context budget, attains a Resolve Rate of only 25.00% (Table 2). If weight-level memorization were the dominant signal, we would expect frontier backbones to approach saturation on the canonical Course Labs and Research Evolutions tasks whose source material is most heavily represented on the public web; the observed ceiling is incompatible with that hypothesis.

The reason is structural: long-horizon Resolve Rate is not gated by single-point knowledge recall. Solving a HORIZONFLOW task requires (i) maintaining cross-unit dependency consistency across many edits, (ii) preserving previously satisfied regression obligations as the codebase evolves, (iii) managing context across compaction events without losing earlier decisions, and (iv) routing the test-driven feedback loop along the dependency DAG. None of these capabilities is conferred by having seen the textbook solution to a single Raft round during pre-training. Memorization can at best supply correct *local fragments*; the long-horizon contract is whether those fragments compose under live regression pressure, which is precisely the dimension on which current agents fail.

Future work. A direct ingest-date $\times$ model-cutoff slice, partitioning tasks by whether their source material predates each backbone's training cutoff and reporting Resolve Rate within each partition, would convert the capability-ceiling argument above into a quantitative robustness check. We leave this slice as future work; the current evidence (a 25% ceiling on the strongest configuration) is already inconsistent with memorization-driven inflation.

F Experiment Settings

Models and providers. RQ1 (E1) varies the backbone under a fixed Claude Code harness across eight flagship models: 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)

fixes the backbone to `azure/gpt-5.4-20260305` and varies the harness across Claude Code, Codex, GitHub Copilot, OpenHands, SWE-agent, and mini-SWE-agent. RQ1 (E3) traces within-family scaling for GPT (Codex), Claude Opus (Claude Code), and Qwen (Qwen Code). Backbones are reached through three uniform routes implemented in `long_horizon_bench/agents/`: an Azure OpenAI deployment with AD-token auth (`api-version 2025-04-01-preview`), the GitHub Copilot token-exchange endpoint, and a DashScope Anthropic-shape proxy. Anthropic-native models go directly through the Anthropic SDK.

Decoding parameters. All agents inherit each provider's defaults except where the harness explicitly overrides them. For the Anthropic SDK path used by Claude Code we set `max_tokens=128,000` and leave temperature/top-p at their SDK defaults; SWE-agent's LiteLLM call has `temperature=0` (top-p disabled because Anthropic rejects both); mini-SWE-agent forwards `model.model_kwargs.reasoning_effort=high` to providers that accept it. No agent uses a custom system prompt beyond the benchmark contract in Appendix P.

Compute budget. The harness imposes a per-task wall-clock budget of $T_{\max}$ chosen per task ($\geq$2 h, capped at 24 h for the longest PR-Sequence and Research-Evolution tasks) and forwards it to each agent as `timeout_sec`. For agents with their own iteration cap we set it well above the budget so wall-clock dominates: Claude Code SDK `max_iterations=10,000`; mini-SWE-agent `-cost-limit 0` (uncapped) with `-exit-immediately`. A single bash tool call inside the container is independently capped at 600 s. Trials run concurrently in a `ThreadPoolExecutor`; per-trial credentials are passed as explicit kwargs (rather than environment mutation) to avoid cross-trial races on auth tokens.

G Metric Definitions

We formalize the five headline metrics used throughout RQ1. Let a task have tests indexed by i and evaluation checkpoints indexed by $\tau \in \{1, \ldots, T\}$. Let $s_i^{(\tau)} \in \{0, 1\}$ denote the pass/-fail outcome of test i at checkpoint τ, and let O_τ denote the set of *regression obligations* active at checkpoint τ (tests previously disclosed and passed at some earlier checkpoint). The full test-state trajectory matrix is $S = [s_i^{(\tau)}]$. Let D_τ denote the set of tests disclosed by checkpoint τ, and let $T_{\max}$ denote the wall-clock time budget. The disclosed-test pass rate at checkpoint τ is

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

G.1 Resolve Rate (RR)

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

Whether every disclosed test passes at task end, including all regression obligations.

G.2 Test Pass Rate (TPR)

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

Average fraction of disclosed tests passing at task end. Summarizes both intermediate progress and retention of earlier commitments in a single number.

G.3 Dependency Depth (Depth)

Depth measures how far the agent advances along the task DAG *within a single invocation*, before any ralph-loop restart hands it a freshly re-derived residual. Let each disclosed test i inherit the topological layer of the development unit it adjudicates, $\mathrm{layer}(i) \in \{0, \ldots, L\}$, where $L = \max_{v \in V} \mathrm{layer}(v)$ is the longest prerequisite chain of the task DAG. Let $\tau^\dagger$ denote the last checkpoint of the first ralph round, that is, the final snapshot before the outer loop, if any, restarts the agent under a re-derived residual. Let $P_{\tau^\dagger} = \{i : s_i^{(\tau^\dagger)} = 1\}$ denote the tests passing at $\tau^\dagger$. Depth is the deepest layer at which every prerequisite test still passes at $\tau^\dagger$, normalized by L:

$$\mathrm{Depth} \;=\; \mathbb{E}_{\text{task}}\left[\frac{\max\{\ell : \forall i,\, \mathrm{layer}(i) \leq \ell \Rightarrow i \in P_{\tau^\dagger}\}}{L}\right]. \qquad (4)$$

Depth isolates the model's intrinsic long-horizon reach from the harness's outer-loop recovery. RR couples both: a task can reach $\mathrm{RR} = 1$ either by a high Depth on the first invocation or by a low Depth that the ralph loop rescues across rounds. Reading Depth alongside RR therefore separates two regimes that RR alone conflates, an early stall ($\mathrm{Depth} \ll 1$, the agent never clears the entry layers on its own) versus a late-graph collapse (high

Depth, RR close to zero, the chain breaks near the end and the outer loop fails to recover).

G.4 Regression Rate (Reg)

A *regression event* occurs whenever a previously passing obligation test fails at a later evaluation point. Checkpoints τ are the inner-loop snapshots emitted by the dual-container watcher (Section 2.6): whenever the agent's cumulative diff against the snapshot baseline exceeds a fixed line threshold, the watcher container freezes the workspace and runs the full disclosed-test suite, recording per-test pass/fail in `observation_history.jsonl` alongside the `newly_passed` and `newly_failed` deltas relative to the previous snapshot. Let τ^- denote the immediately preceding triggered snapshot at which test i was evaluated, let O_τ denote the obligation set at τ (the union of all tests that were `newly_passed` at any earlier triggered snapshot), and define

$$R = \#\{(i,\tau) : i \in O_\tau,\ s_i^{(\tau^-)}=1,\ s_i^{(\tau)}=0\}. \tag{5}$$

Then

$$\text{Reg} = \frac{R}{\sum_\tau |O_\tau|}. \tag{6}$$

Equivalently: how often the agent breaks something it had already gotten right.

Reg should be read jointly with Resolve Rate. Backbones with very low Resolve Rate rarely satisfy any obligation in the first place, so $|O_\tau|$ stays small and Reg trends mechanically toward zero; this is a denominator effect, not stability. Conversely, a backbone with moderate RR but high Reg (e.g., DeepSeek-V4P at RR=11.61%, Reg=16.60% in Table 3) is best understood as "capable but unstable". We therefore use Reg as a diagnostic rather than as a standalone ranking signal.

G.5 Tokens (Tok)

Total tokens consumed per task, summed across input and output across all checkpoints. Captures the compute cost of producing the above outcomes.

H Model Release Dates and Parameter Scales

To make the within-family scaling in Table 2 (right, mint) interpretable, we tabulate the release date and reported parameter scale of every backbone evaluated in RQ1. The three families do not share a common starting point: the GPT family spans `gpt-4o` (May 2024) through `gpt-5.5` (released 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 with very different release cadences, the per-family monotonicity in Resolve Rate should be read as a generational delta rather than a parameter-count scaling law.

Table 7: Backbones evaluated in RQ1. OW = open-weight; "undisclosed" = closed-weight checkpoints whose providers have not published a parameter count. Release dates are the public-availability dates of the checkpoint version we evaluated.

Model	Family	Release	Params	Ref.
GPT-4o	GPT	2024-05	undisclosed	(OpenAI, 2024a)
GPT-5	GPT	2025-08	undisclosed	(OpenAI, 2025b)
GPT-5.2	GPT	2025-11	undisclosed	(OpenAI, 2025c)
GPT-5.4	GPT	2026-03	undisclosed	(OpenAI, 2026a)
GPT-5.5	GPT	2026-04	undisclosed	(OpenAI, 2026b)
Sonnet-4	Claude	2025-05	undisclosed	(Anthropic, 2025a)
Sonnet-4.5	Claude	2025-09	undisclosed	(Anthropic, 2025d)
Opus-4.5	Claude	2025-11	undisclosed	(Anthropic, 2025c)
Opus-4.7	Claude	2026-02	undisclosed	(Anthropic, 2026)
Qwen2.5-72B	Qwen	2024-09	72 B (OW)	(Qwen Team, 2024)
Qwen3-Max	Qwen	2025-09	undisclosed	(Alibaba Cloud, 2025)
Qwen3.5-Plus	Qwen	2025-12	undisclosed	(Alibaba Group, 2026)
Qwen3.6-Plus	Qwen	2026-03	undisclosed	(Qwen Team, 2026)
Gemini-3.1-Pro	Gemini	2026-03	undisclosed	(Google DeepMind, 2026)
GLM-5.1	GLM	2026-02	undisclosed	(Z.AI, 2026)
DeepSeek-V4P	DeepSeek	2026-01	undisclosed	(DeepSeek-AI, 2026)
Kimi-2.6	Moonshot	2026-03	undisclosed	(Moonshot AI, 2026)
Grok-4.1-FR	xAI	2026-02	undisclosed	(xAI, 2026)

I Pass@K for the Within Family Scaling Panel

The monotonic Resolve Rate trend across model generations within each family, reported in the left panel of Table 2, rests on twelve single trial measurements and is therefore vulnerable to sampling noise. To assess robustness, we additionally evaluate pass@3 on the same generations and report it here. The within family ordering is preserved at $K>1$ for every family, so the headline trend in Table 2 is not an artifact of single trial noise.

J Planning-DAG Metrics

We formalize the six planning-DAG metrics used in RQ2 (Table 3). Let $G_A = (V_A, E_A)$ denote the agent-generated 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 the set of units present in both DAGs.

Table 8: **pass@K Resolve Rate (%) for the within family scaling panel.** Each cell reports the fraction of tasks resolved by at least one of K independent trials.

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
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
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

J.1 Edge F1

Let $E_\cap = E_A \cap E_H$, and define

$$P_{\text{edge}} = \frac{|E_\cap|}{|E_A|}, \qquad R_{\text{edge}} = \frac{|E_\cap|}{|E_H|}. \tag{7}$$

$$\text{Edge F1} \;=\; \frac{2\,P_{\text{edge}}\,R_{\text{edge}}}{P_{\text{edge}} + R_{\text{edge}}}. \tag{8}$$

Exact recovery of direct prerequisite edges. Low precision indicates hallucinated dependencies; low recall indicates missing prerequisites that risk out-of-order implementation.

J.2 TC-F1

Let $\mathrm{TC}(\cdot)$ denote the transitive closure of a DAG's edge set, and write $T_\cap = \mathrm{TC}(E_A) \cap \mathrm{TC}(E_H)$. Define

$$P_{\text{tc}} = \frac{|T_\cap|}{|\,\mathrm{TC}(E_A)|}, \qquad R_{\text{tc}} = \frac{|T_\cap|}{|\,\mathrm{TC}(E_H)|}. \tag{9}$$

$$\text{TC F1} \;=\; \frac{2\,P_{\text{tc}}\,R_{\text{tc}}}{P_{\text{tc}} + R_{\text{tc}}}. \tag{10}$$

A more permissive measure of ordering agreement: it credits an agent that captures u *before* v via a different intermediate path than the human development DAG.

J.3 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 nodes with no prerequisites; later layers depend only on earlier ones).

$$\text{Layer}\,\rho \;=\; \text{Spearman}\big(\{\ell_A(v)\}_{v\in V_\cap}, \{\ell_H(v)\}_{v\in V_\cap}\big). \tag{11}$$

Captures whether agents place units at similar development *stages* as the human reference, independent of exact edge structure.

J.4 Critical Path Ratio

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

$$\mathrm{CPR}(G) \;=\; \frac{\mathrm{CP}(G)}{|V|}. \tag{12}$$

Values near 1 indicate a near-linear chain (over-serialization); lower values indicate that the agent admits parallel branches. We also report a human-normalized variant $\mathrm{CPR}(G_A)/\,\mathrm{CPR}(G_H)$ in extended results.

J.5 Width Ratio

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

$$\mathrm{WR} \;=\; \frac{w(G_A)}{w(G_H)}. \tag{13}$$

Values close to 1 indicate that the agent preserves the human development DAG's concurrent ready-frontier width; <1 indicates over-serialization, >1 over-parallelization that may have dropped real dependencies.

J.6 Branch Switching Rate

Assign each unit v a branch identifier $b(v)$ corresponding to its earliest reachable source node in G_H (units with multiple source ancestors are labeled *mixed*). Given the agent's planned (or executed) implementation order $\sigma = (v_{\pi(1)}, \ldots, v_{\pi(m)})$, let $K = |\{\, i : b(v_{\pi(i)}) \neq b(v_{\pi(i+1)})\,\}|$.

$$\mathrm{BSR} \;=\; \frac{K}{m-1}. \tag{14}$$

High values indicate breadth-first frontier traversal across branches; low values indicate depth-first completion of one branch before switching. Neither direction is universally preferred; we interpret the value relative to the shape of G_H.

K Extended RQ2 Grid

Metric definitions for Table 3. *Planning fidelity.* **Edge F1** = F1 between agent and human development DAG edge sets. **Layer** ρ = Spearman correlation between agent and human topological layer indices. **CPR** = critical-path length of

G_A divided by $|V_A|$ (raw, not normalized to gold). **WR** = mean ready-frontier width of G_A divided by that of G_H. *Solution* (agent-correct units only). **PatchLen** = patch lines per matched unit divided by gold patch lines. **Jacc** = token-level Jaccard distance between agent and gold patch. *Testing.* **#T** = number of agent-authored tests per task. **F2P** = fraction of agent-authored tests that fail on the agent's initial state and pass on the gold solution. *Ralph loop* (per task, averaged). **Rounds** = mean number of outer-loop iterations the harness invokes before terminating, fitted on started trials with a per-cell p_{95} cap to suppress single-trial blow-ups; under our outer loop, a new session inherits the prior round's residual whenever `progress_percent` < 100, so harnesses without effective in-session compaction accrue more rounds as the residual fails to clear in one window. **ΔTPR$_r$** = mean change in disclosed-test pass rate (percentage points) across each round transition, measured as TPR at the start of round $k+1$ minus TPR at the end of round k from the dual-container watcher's `observation_history.jsonl`; negative values indicate test loss accompanying outer-loop hand-off. *Compaction* (per task, averaged). **Ev/run** = mean number of compaction events fired per task, detected from the trajectory (Appendix M). **ΔTPR$_c$** = mean change in disclosed-test pass rate (percentage points) in a ± 60 s window around each compaction event; negative values indicate test loss accompanying window compression.

Pruned planning metrics. The main paper's Table 3 reports a pruned grid covering planning, solution, testing (RQ2) and loop/compaction dynamics (RQ3). We move four planning metrics here that are either highly correlated with a column already shown or have a direction of preference that depends on the dependency DAG and would mislead a reader scanning the main table.

TC-F1 (planning). Transitive-closure F1 (Appendix J.2) is strongly correlated with Edge F1 (Appendix J.1) in our runs ($\rho > 0.9$ across the six harnesses), so it adds little ranking information beyond Edge in the main table. We retain it here because its absolute level still calibrates how well an agent recovers *indirect* prerequisite structure.

Branch Switching Rate (planning). BSR (Appendix J.6) does not have a universally preferred direction: a high BSR is desirable when G_H exposes many simultaneously ready frontiers and undesirable when G_H is a long sequential chain. Including it as a leaderboard column in the main paper would conflict with the prose conclusion that scaffolds collapse the plan into a near-linear chain.

Test-Ratio (testing). The main table reports #T (raw test count). Test-Ratio is exactly $\#T/\#T_{\text{native}}$ with $\#T_{\text{native}} = 44$, so it is fully derivable from the #T column.

JSD (testing). Per-unit count distribution divergence between agent and native test placement. We additionally plan to replace JSD with a unit-position histogram (see `test-distribution` todo) since divergence alone does not reveal whether tests concentrate on early or late units.

Where the extra patch lines come from. The main text reports that PatchLen exceeds gold while Jacc to gold stays at a comparable moderate level. Inspecting the diffs explains both signals at once. Agents adopt a more uniform phrasing and attach verbose and often unnecessary comments to the patch, and a substantial share of the extra lines comes from rewriting blocks that already exist in the repository because the agent failed to retrieve the relevant context and re derived the file from scratch. Length therefore grows while the token vocabulary stays close to the human reference, which is why Jacc does not move in step with PatchLen.

L Per-Source Resolve Rate Breakdown

Section 3.2 treats the 112 HORIZONFLOW tasks as a single pool when reporting aggregate Resolve Rate and planning metrics. The three task sources (PR Sequences, Course Labs, Research Evolutions) however carry different ground-truth strength: PR Sequences inherit a fully source-evidenced dependency DAG from merged-commit history, Course Labs inherit a module-level dependency structure from the curriculum, and Research Evolutions inherit a coarser successor-relation chain from the citation graph. Averaging across the three sources can therefore mask qualitatively different failure modes.

Table 9 reports per-source Resolve Rate for every (agent, backbone) configuration evaluated in RQ1. The aggregate column reproduces the headline number from Table 2 and equals $\text{RR}_{\text{agg}} = \frac{1}{112}\big(29 \cdot \text{RR}_{\text{PR}} + 57 \cdot \text{RR}_{\text{Course}} + 26 \cdot \text{RR}_{\text{Research}}\big)$ up to rounding.

Two findings hold uniformly across configurations and merit attention before reading the per-

source columns.

Finding 1: PR Sequences are uniformly the hardest source. Across all 14 (agent, backbone) configurations, per-source Resolve Rate on PR Sequences stays at most a few percent, and in 9 of 14 configurations no PR Sequence task resolves end to end. The PR Sequences pool is dominated by merge chains from actively maintained repositories where each segment depends on cross-module invariants established several PRs earlier; once an agent breaks a prerequisite invariant in an early segment, the rest of the chain regresses irrecoverably. This pattern is consistent with the regression-after-compaction story in RQ3 but is in fact visible even without any compaction event firing.

Finding 2: Course Labs absorb most of the resolved tasks. For every configuration, Course Labs supply $\geq 70\%$ of the resolved-task count. Course Labs have shallower dependency depth on average (median depth 3 vs. 6 for the pool) and a curriculum-imposed module decomposition that aligns with the human development order an agent would re-derive from first principles. Per-source Resolve Rate on Course Labs is therefore the column most sensitive to backbone capability and tracks the aggregate RR ranking most closely.

Research Evolutions sit in between: the moderate pool size (26 tasks) gives a more stable per-source rate than in earlier preview cuts, but the strongest configurations resolve at most a handful of tasks while the weakest resolve none. The pattern of per-source spread matters as much as the column-level rate at this sample size.

Two implications follow for how the aggregate HORIZONFLOW Resolve Rate should be read. First, comparing two agents on a single aggregate number conflates capability differences on the Course-Labs pool with their (universally poor) PR-Sequences performance, since the latter contributes little spread. The cross-agent gap in Table 2 bottom-left is therefore largely a Course-Labs gap. Second, future work that improves PR-Sequences performance specifically should be reported on the per-source breakdown rather than as a small delta on the aggregate, because a doubling of PR-Sequences RR moves the aggregate by less than one percentage point.

M Compaction Mechanism Internals

We document the in-the-wild compaction mechanism of every harness in RQ3 along three axes: *trigger rule* (the condition under which a compaction event fires), *retention ratio* (what fraction of pre-event tokens survives the event), and *post-event visibility* (which earlier turns the model can still attend to after compaction).

Claude Code SDK auto-compaction. The Anthropic Agent SDK fires an internal summarization pass when accumulated context crosses a provider-set threshold (approximately 80% of the model's effective window). The mechanism replaces the prefix of the trajectory with a model-authored summary message and retains the most recent tool-call exchanges verbatim; earlier turns are no longer attendable except through the summary.

Sliding-window truncation (Codex, mini-swe-agent). A fixed-size FIFO queue over turns: once the queue length exceeds the configured cap, the oldest turn is dropped without summarization. Retention ratio is exactly $1 - 1/N$ at each event; visibility of dropped turns is zero.

Memory pruning (OpenHands). A scoring-based eviction over the event log: low-utility events (read-only observations, redundant tool outputs) are pruned first; the remaining log is presented intact. Retention ratio is variable and event-driven.

External-file scratchpad (plan.json). Not an in-context compaction at all: the agent serializes salient state to `/workspace/agent_plans/plan.json` and re-reads it on demand. The model's context never compacts in the strict sense, but the file fetch occupies tokens at re-read time, which we account for separately as recovery work.

M.1 Loop and Compaction Metric Definitions

We formalize the four loop and compaction columns reported in Table 3. For a single trial, let the outer ralph loop emit rounds indexed $k \in \{1, \ldots, K\}$, recorded in `outer_loop_history.jsonl` with start timestamp t_k^{R}. The dual-container watcher writes a snapshot to `observation_history.jsonl` whenever the cumulative diff against the snapshot baseline exceeds a fixed line threshold, yielding a sequence of triggered observations $\{(t_\tau, p_\tau, f_\tau)\}_\tau$ where p_τ and f_τ are the passed and failed disclosed-test counts at snapshot τ. Define the disclosed-test pass rate at snapshot τ as $\mathrm{TPR}(t_\tau) = p_\tau/(p_\tau + f_\tau)$, with $\mathrm{TPR}(\cdot)$ extended

Table 9: **Per-source Resolve Rate.** Each row reports the fraction of resolved tasks restricted to one of the three sources, plus the pool aggregate (Table 2). PR Sequences are uniformly the hardest source; Course Labs absorb most of the resolved count for every configuration; Research Evolutions are reported at n=26.

Agent	**Backbone**	**PR Seq.** (n=29)	**Course** (n=57)	**Research** (n=26)	**All** (n=112)
Cross-backbone under a fixed Claude Code agent					
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%	19.30%	11.54%	12.50%
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%	7.02%	3.85%	4.46%
Claude Code	Grok-4.1-FR	0.00%	5.26%	0.00%	2.68%
Cross-agent under a fixed `gpt-5.4` *backbone*					
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%

to arbitrary timestamps via left-continuous step interpolation.

Rounds.

$$\text{Rounds} \;=\; \mathbb{E}_{\text{trial}}[\,K\,]\,. \tag{15}$$

Mean number of outer-loop iterations the harness invokes per task, taken over all trials (including those that terminate early in failure modes `agent_timeout`, `agent_rate_limited`, `unknown_agent_error`, since premature termination is itself a property of the loop controller).

ΔTPR$_r$ (test loss per round transition). For each round transition $k \to k{+}1$ within a trial, let $\mathrm{TPR}(t_k^{\mathrm{R}} - \epsilon)$ denote the disclosed-test pass rate at the last triggered snapshot strictly before t_{k+1}^{R}, and $\mathrm{TPR}(t_{k+1}^{\mathrm{R}} + \epsilon)$ the rate at the first triggered snapshot at or after t_{k+1}^{R}. Then

$$\begin{aligned}\Delta\mathrm{TPR}_r \;=\; \mathbb{E}_{\text{trial},\,k}\big[\,&\mathrm{TPR}(t_{k+1}^{\mathrm{R}}{+}\epsilon) \\ &-\; \mathrm{TPR}(t_k^{\mathrm{R}}{-}\epsilon)\,\big] \times 100\,\text{pp}.\end{aligned} \tag{16}$$

Reported in percentage points. Negative values indicate test loss accompanying outer-loop handoff; trials with $K = 1$ contribute zero transitions.

Ev/run. Let E denote the number of compaction events detected in a trial's `claude_sdk_trajectory.jsonl`. Then

$$\text{Ev/run} \;=\; \mathbb{E}_{\text{trial}}[\,E\,]\,. \tag{17}$$

The detection heuristic is harness-specific and documented in Appendix M.2.

ΔTPR$_c$ (test loss per compaction event). For each detected event at timestamp t^{C}, let

$$\begin{aligned}\overline{\mathrm{TPR}}^{\,-}(t^{\mathrm{C}}) &= \mathrm{mean}\big\{\mathrm{TPR}(t_\tau)\;: \\ &\qquad t^{\mathrm{C}} - 60\mathrm{s} \le t_\tau < t^{\mathrm{C}}\big\}, \end{aligned} \tag{18}$$

$$\begin{aligned}\overline{\mathrm{TPR}}^{\,+}(t^{\mathrm{C}}) &= \mathrm{mean}\big\{\mathrm{TPR}(t_\tau)\;: \\ &\qquad t^{\mathrm{C}} < t_\tau \le t^{\mathrm{C}} + 60\mathrm{s}\big\}. \end{aligned} \tag{19}$$

Then

$$\begin{aligned}\Delta\mathrm{TPR}_c \;=\; \mathbb{E}_{\text{trial},\,t^{\mathrm{C}}}\big[\,&\overline{\mathrm{TPR}}^{\,+}(t^{\mathrm{C}}) \\ &-\; \overline{\mathrm{TPR}}^{\,-}(t^{\mathrm{C}})\,\big] \times 100\,\text{pp}.\end{aligned} \tag{20}$$

Reported in percentage points. Events with no triggered snapshot in one of the two windows are dropped. The ± 60 s window is chosen so that, at the watcher's nominal 15 s polling cadence, both halves contain at least two snapshots in expectation, attenuating shot noise from any single observation.

Reading the two ΔTPR columns. $\Delta\mathrm{TPR}_r$ and $\Delta\mathrm{TPR}_c$ are not redundant. $\Delta\mathrm{TPR}_r$ probes the cost of the harness handing control back to the model under a freshly re-derived residual, where pass-rate loss reflects the residual losing track of previously satisfied obligations; $\Delta\mathrm{TPR}_c$ probes the cost of in-context compression of a single growing transcript, where pass-rate loss reflects the summary or pruning dropping prerequisite invariants. Harnesses heavy on one mechanism are light on the other (Appendix M.3), and the two columns together separate *residual drift* (large $|\Delta\mathrm{TPR}_r|$, small $|\Delta\mathrm{TPR}_c|$) from *window collapse* (small $|\Delta\mathrm{TPR}_r|$, large $|\Delta\mathrm{TPR}_c|$).

M.2 Compaction Event Detection

The SDK trajectories we record do not carry an explicit `event=compaction` marker, so E is recovered post hoc from the per-turn usage block. Across all harnesses, a compaction event is defined as a turn i at which $u_i^{\text{in}} \leq 0.7 \cdot u_{i-1}^{\text{in}}$, where u_i^{in} is the assistant turn's reported input-token count (cache-read tokens included). Consecutive turns satisfying the rule are collapsed into a single event timestamped at the first qualifying turn.

M.3 Coupling Between the Two Layers

N Compaction Counterfactual Configurations

To separate "compaction causes regressions" from "compaction co-occurs with hard task phases", we re-run two harnesses on the same (model, task) pairs under two compaction settings each. The backbone is held at `gpt-5.4` for Codex and Opus-4.7 for Claude Code SDK; tasks are sampled to span all three sources.

Claude Code SDK. Aggressive setting: trigger threshold lowered to 50% of the effective window. Delayed setting: threshold raised to 95%. All other SDK parameters held at defaults.

Codex. Aggressive setting: sliding-window cap halved relative to the default. Delayed setting: cap doubled.

For each (harness, setting) pair we report RR, Reg, mean events per run, and post-event ΔReg, paired by task.

O Harness Adaptations

HORIZONFLOW wraps six off-the-shelf agents behind a uniform `BaseAgent` contract. The wrappers normalize three things across agents, namely credential plumbing, model identifiers, and the workspace contract, so that the only varying surface across rows of Table 2 is the agent itself.

Workspace and git contract. Every agent is dropped into `/workspace` inside a per-task Docker container built from the task's `Dockerfile`. The directory is initialized as a git repo with three sentinel subtrees the harness watches: `requirements/` (per-unit YAML manifests, all visible to the agent from the start), `agent_plans/` (optional `plan.json` for RQ2 plan analysis), and `requirement_patches/` (one `<slug>.diff` per implemented unit, written by the agent with `git diff -cached`). Patch files MUST be non-empty for a unit to count as complete; checkpoint detection (`long_horizon_bench/harness/regression_harness.py`) uses `min_diff_lines=5` as the minimum diff growth that may trigger a snapshot, with the per-unit threshold derived from the smallest gold artifact still on the ready frontier.

Per-agent adapters. *Claude Code* runs through the Anthropic Agent SDK with a `ContainerWorkspaceBackend` that proxies the SDK's bash/edit tools into the task container; trajectories and per-trial usage land at `claude_sdk_trajectory.jsonl` and `claude_sdk_result.json`. *Codex* is patched to use a custom `model_provider` block (`lhb_azure` or `lhb_proxy`) injected into `config.toml`, since stock Codex requires the Responses wire protocol. *GitHub Copilot CLI* runs in non-interactive mode against a sidecar (`AZURE_SIDECAR_PORT=23337`) reached via the container's default-gateway IP (no `host.docker.internal` in the default compose). *Cursor CLI* maps Anthropic shorthand to Cursor's own model IDs (`anthropic/claude-opus-4.7` $\rightarrow$ Cursor's identifier) before invoking `agent -model`. *mini-SWE-agent* is installed inside the container at `/opt/python` via `pip -prefix` with an explicit `PYTHONPATH` fix-up; `copilot/`-prefixed models are rewritten to `openai/` so LiteLLM hits the OpenAI-compatible proxy. *SWE-agent*'s LiteLLM call is sed-patched to drop the `top_p` keyword (Anthropic rejects requests carrying both `temperature` and `top_p`, and SWE-agent's defaults set both).

Auth race fix. Earlier versions of the wrapper pushed Azure AD tokens and Copilot keys into `os.environ` inside a context manager. Under the harness's `ThreadPoolExecutor` this raced: two trials would overwrite each other's tokens, and one trial's `finally` could clear the variable mid-call. The current adapters pass credentials as explicit kwargs to `run_task` (`azure_ad_token_provider`, `copilot_api_key`, `anthropic_api_key`), eliminating the global-state contention.

P Prompt Configuration

The benchmark contract is composed by `long_horizon_bench/harness/instruction_`

composer.py. Each agent receives the task instruction concatenated with the *Benchmark Contract* (always), the *DAG Plan Requirement* (RQ2 plan-fidelity runs only), and an *Agent Profile Addendum* selected by agent class. The Oracle agent bypasses composition.

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 (replaces the optional planning paragraph for RQ2 plan-fidelity 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, GitHub Copilot, and Cursor, no profile addendum is appended; they receive only the Benchmark Contract (and the DAG-plan replacement when applicable).

Q Harness Runtime Configuration

The HORIZONFLOW evaluation harness is invoked via `lhb run -config config.yaml`. All parameters can be set in the YAML configuration file or overridden by CLI flags. Table 10 summarises the complete parameter space.

Table 10: Runtime parameters for `lhb run`. Defaults are shown in parentheses; – indicates no default (required or derived from context).

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).
Agent	`agent`	`oracle`	Built-in agent name.
	`agent_import_path`	–	Import path for a custom agent class.
	`model`	–	LLM in `provider/model` format.
	`agent_kwargs`	`{}`	Key–value pairs forwarded to agent.
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 agent 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 agent budget (seconds).
	`global_test_timeout_sec`	per-task	Override test budget (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 scaling without editing individual task files.

Outer-loop structure. The harness executes an *outer loop* around the agent, where each round is a fresh shell session with the workspace persisted from prior rounds. The number of rounds is controlled by `outer_loop_count` (default 3); setting it to `0` or `unlimited` removes the cap, letting the agent iterate until it exhausts the timeout budget. 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).

Model configuration. API keys 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. Keys already set in the process environment are preserved unless `override_process_env: true`.

Example configuration. The following shows a minimal configuration used in our main evaluation runs.

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-..."
```