DR4CODE-BENCH: BENCHMARKING RESEARCH EXPERIENCE REUSE ACROSS SOFTWARE EVOLUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Related changes in evolving software repositories often revisit the same concepts, creating opportunities to reuse earlier repository investigation. Whether such reuse improves coding success, reduces repeated investigation, or both requires direct evaluation. We introduce **DR4Code-Bench**, an executable benchmark for evaluating pre-solution Investigation Report reuse across concept-linked software evolution tasks. It contains 88 executable tasks from 10 Python repositories organized into 22 concept-propagation chains. Each task is independently reconstructed and validated. Agents investigate the current repository before coding; only their Investigation Reports are carried across tasks, while prior agent patches, research traces, and oracle feedback are withheld. We evaluate six models on the 66 non-anchor tasks under No-History, Full-History, Recent-1, and Top-1 conditions, using a shared research-to-coding protocol and scoped outcome evaluator. Single-run correctness differences vary across models and policies. Three additional runs for DeepSeek-V4-Flash and Qwen3.8-Flash show that the direction of observed correctness gains and the strongest fixed policy can change across executions. In the six-model single-run evaluation, the reuse policies require 2.1–2.8 fewer research calls per episode on average across models and tasks, with small mean changes in coding calls. The reductions persist when comparisons are restricted to paired episodes in which research completes under both conditions. Repeated-run analyses show more consistent research-call reductions for DeepSeek than for Qwen. These results distinguish savings in repository investigation from improvements in final coding correctness and establish DR4Code-Bench as a setting for measuring both. [1]

1 INTRODUCTION

Software changes often require coding agents to investigate repository-specific contracts, dependencies, and conventions before implementing the requested behavior (Chen et al., 2025; Zhang et al., 2026). In evolving repositories, development proceeds through sequences of changes over time (Le et al., 2025; Deng et al., 2026). Related changes may revisit the same concepts, creating opportunities to reuse investigation performed for earlier tasks. A report identifying relevant components or explaining their behavioural relationships may help an agent approach a subsequent change without rediscovering the same context. This raises a practical question: *how much value do prior investigation reports retain for later, concept-linked software changes?* We consider two possible benefits: improved coding success and reduced repository-investigation effort.

Evaluating this question requires separating investigation from other forms of problem-solving experience. Repository-level benchmarks such as SWE-bench evaluate tasks as independent episodes and therefore do not directly measure cross-task reuse (Jimenez et al., 2024). Cross-task evaluations such as SWE-ContextBench instead support reuse of completed execution trajectories or their summaries (Zhu et al., 2026). Such experience can combine repository observations with implementation decisions and execution feedback. Building on work that treats repository investigation as a distinct pre-solution stage (Singh et al., 2025), we focus on a narrower transfer artefact: an *Investigation Report* produced before coding that records task-relevant concepts and inspected files.

[1] All benchmark data, code, and evaluation artifacts will be made publicly available upon publication.

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

This allows us to study the utility of prior investigation without transferring earlier agent patches, research traces, or oracle feedback.

The value of this reuse depends on how earlier investigation relates to the requirements and repository state of the current task. Earlier reports may provide useful localisation and behavioural context, repeat information readily recoverable from the current repository, or describe behaviour that later changes have revised. A report may reduce the work needed to locate and understand relevant code while leaving the eventual coding outcome unchanged. The usefulness of report reuse must therefore be assessed through both coding outcomes and investigation effort, rather than inferred from task relatedness alone.

We introduce **DR4Code-Bench**, an executable benchmark containing 88 tasks from 10 Python repositories, organised into 22 concept-propagation chains. The chains link historical changes through evidence of concept use, while each task is independently reconstructed and validated. Agents solve each task through a research-to-coding episode: they investigate the current repository, produce an Investigation Report, and then implement the requested change. Only reports cross task boundaries. Independent reconstruction prevents earlier agent implementations from altering task states, while report-only transfer separates prior investigation from completed solution artefacts and evaluation feedback. This design enables us to measure how access to prior Investigation Reports affects coding success and repository-investigation effort.

We evaluate six models on the 66 non-anchor tasks under No-History, Full-History, Recent-1, and Top-1 conditions. Single-run correctness differences vary across models and policies. Three additional end-to-end runs for DeepSeek-V4-Flash and Qwen3.8-Flash show that the direction of observed gains and the strongest fixed policy can change across executions. Investigation effort shows a clearer aggregate pattern: in the six-model single-run evaluation, the reuse policies use 2.1–2.8 fewer research calls per episode on average across models and tasks, with small mean changes in coding calls. Repeated-run analyses show more consistent research-call reductions for DeepSeek than for Qwen. These findings distinguish savings in repeated investigation from improvements in final correctness and motivate evaluating the two outcomes separately.

We make three primary contributions.

- **A concept-linked executable benchmark.** DR4Code-Bench comprises 88 independently reconstructed and validated tasks from 10 real Python repositories, organised into 22 chains linking historical changes through evidence of concept use.
- **An isolation protocol for investigation reuse.** Our research-to-coding protocol restricts cross-task transfer to Investigation Reports, separating this information source from prior agent patches, research traces, and oracle feedback.
- **An empirical characterization of reuse utility.** We evaluate coding outcomes and resource use across six models and examine their stability through additional runs for two models, distinguishing investigation savings from execution-sensitive correctness gains.

2 Related Work

2.1 Coding Benchmarks and Repository Investigation.

Early code-generation benchmarks such as HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) evaluate functional correctness on self-contained programming problems, while LiveCodeBench (Jain et al., 2025) extends this setting with continuously updated problems and broader code-related capabilities. SWE-bench (Jimenez et al., 2024) moves evaluation to real repository-level issue resolution, where agents must reason over an existing codebase rather than synthesise an isolated program. A related line of work studies the investigation required before implementation. AutoCodeRover (Zhang et al., 2024) performs iterative structure-aware code search, LocAgent (Chen et al., 2025) uses repository structure for multi-hop localisation, and SWE-Explore (Zhang et al., 2026) evaluates repository exploration separately from final patch generation. Code Researcher (Singh et al., 2025) most directly treats repository investigation as a distinct pre-solution stage, gathering context from code and commit history before synthesis. More broadly, DeepResearch Bench (Du et al., 2025) characterises deep research as iterative evidence gathering and synthesis. These works motivate explicit evaluation of investigation and its outputs; we exam-

ine how pre-solution Investigation Reports affect coding success and investigation effort on later concept-linked tasks.

2.2 Agent Memory and Cross-Episode Experience Reuse.

Agent-memory benchmarks study how information accumulated over time can be retained, retrieved, updated, and reused. LongMemEval (Wu et al., 2025) evaluates long-term interactive memory across multi-session and temporally changing information. MemoryAgentBench (Hu et al., 2025) broadens this perspective to retrieval, test-time learning, long-range understanding, and selective forgetting, while MemBench (Tan et al., 2025) distinguishes factual and reflective memory and evaluates effectiveness, efficiency, and capacity. More recent settings emphasise memory for future action: MemoryArena (He et al., 2026) studies interdependent multi-session tasks, and EvoMemBench (Wang et al., 2026) organises memory along within- versus cross-episode and knowledge- versus execution-oriented dimensions. In software engineering, SWE-Bench-CL (Joshi et al., 2025) organises repository-specific task sequences using chronology and difficulty-based curricula, while SWE-ContextBench (Zhu et al., 2026) evaluates reuse of completed problem-solving trajectories and their summaries across related coding tasks. Our distinction concerns the source and timing of the transferred information: Investigation Reports are generated before implementation, rather than summarised from completed problem-solving trajectories. DR4Code-Bench evaluates this form of report reuse without transferring prior agent patches, research traces, or oracle feedback.

2.3 Software Evolution Benchmarks.

Recent benchmarks moved beyond isolated issue resolution toward longer-horizon software evolution. SWE-EVO (Le et al., 2025) constructs release-level evolution tasks that require coordinated repository changes, while SWE-Milestone (Deng et al., 2026) reconstructs continuous streams of historical development milestones. These benchmarks evaluate agents' ability to carry out extended software changes, whereas DR4Code-Bench evaluates report reuse across independently reconstructed tasks. Concept links organise these tasks into related sequences, while Investigation Reports are the only artefacts transferred between episodes.

Together, these research directions motivate evaluating the utility of pre-solution Investigation Reports across concept-linked software evolution tasks. DR4Code-Bench combines an explicit pre-solution reporting stage, report-only cross-task transfer, and independent task reconstruction to measure changes in coding success and investigation effort.

Table 1 summarises the resulting distinction across six design dimensions.

Table 1: Comparison of design features included in the reported evaluation settings of related coding-agent systems and benchmarks (✓: included; ×: not included; –: not applicable). *Pre-solution research* isolates repository investigation from implementation; *Pre-solution-only memory* restricts cross-task transfer to repository observations recorded before implementation, excluding prior agent patches, research traces, and oracle feedback. *Software evolution* explicitly incorporates chronological relationships between repository changes or release-to-release transitions into task construction.

	Investigation and memory			**Task structure**		
Work	**Pre-solution research**	**Pre-solution-only memory**	**Cross-task memory**	**Task chains**	**Software evolution**	**Linkage**
SWE-Explore	✓	–	×	×	×	–
Code Researcher	✓	–	×	×	×	–
SWE-Bench-CL	×	×	✓	✓	✓	Time/difficulty
SWE-ContextBench	×	×	✓	✓	×	Issue/PR
SWE-EVO	×	–	×	×	✓	–
SWE-Milestone	×	×	✓	✓	✓	DAG
DR4Code-Bench	✓	✓	✓	✓	✓	**Concept**

3 DR4CODE-BENCH

DR4Code-Bench links independently reconstructed historical coding tasks into concept-propagation chains and evaluates cross-task reuse of Investigation Reports while withholding earlier code and oracle feedback (Figure 1).

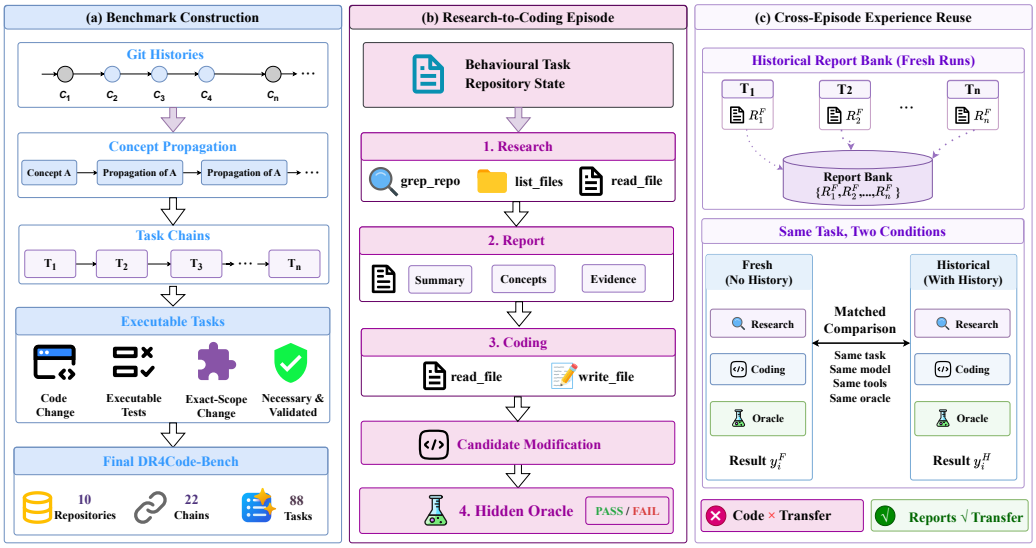


Figure 1: Overview of DR4Code-Bench construction and evaluation. Historical commits are linked into concept-propagation chains and converted into validated exact-scope tasks. Each task is then solved through a research-to-coding episode, in which Investigation Reports from earlier tasks may be reused without transferring their code changes or oracle feedback.

3.1 TASK FORMULATION

For a task derived from target commit c_i, let S_i denote the complete repository snapshot at that commit and τ_i its natural-language behavioural task description. Rather than checking out the parent revision, which may remove unrelated target-state changes, we selectively reverse only the validated task-specific change $\Delta_i = (\Delta_i^{\text{code}}, \Delta_i^{\text{oracle}})$, where Δ_i^{code} is the validated production-code change within the selected function or class scopes, and Δ_i^{oracle} is the corresponding target-oracle change. Selected production definitions and oracle files are restored to their parent versions, or removed if introduced by c_i, while all other target-state changes are preserved. The resulting agent-visible state is

$$X_i = S_i \ominus \Delta_i, \tag{1}$$

where $\ominus$ denotes this selective reversal.

Let O_i denote the executable target oracle which is hidden during investigation and coding and restored only for evaluation. We therefore define a benchmark task as $T_i = (\tau_i, X_i, O_i)$.

Tasks are organized into chronological concept-propagation chains $C = (T_1, T_2, \ldots, T_n)$, where T_1 introduces the root concept and each later task contains propagation evidence involving a concept introduced earlier in the chain.

3.2 RESEARCH-TO-CODING EPISODE

Each task is solved through a two-phase research-to-coding episode. During research, the agent receives (τ_i, X_i, H_i), where H_i contains historical Investigation Reports from earlier tasks and $H_i = \varnothing$ under the no-history condition. Research is read-only: the agent may search and inspect the repository, while historical reports serve only as leads and the current repository state remains authoritative.

We represent the two phases as

$$(R_i, \Gamma_i) = \mathrm{Research}(\tau_i, X_i, H_i), \quad \widehat{\Delta}_i^{\mathrm{code}} = \mathrm{Code}(\tau_i, X_i, R_i, \Gamma_i), \tag{2}$$

where R_i is the current Investigation Report and Γ_i the complete research trace. The report summarizes task-relevant concepts and inspected files without copying source code, reproducing algorithms, or prescribing a complete patch. During coding, the retained trace remains available; repository-wide search and directory exploration are disabled, although known paths may be inspected and production code modified.

The resulting candidate is restricted to the validated task scope and evaluated with the hidden oracle O_i. Across tasks, only earlier Investigation Reports may enter H_i; code, research traces, and oracle feedback are never transferred.

3.3 Benchmark Construction

We construct DR4Code-Bench from the Git histories of real Python repositories. The construction pipeline has two goals: to identify meaningful histories of concept propagation, and to convert each retained historical change into an independently executable coding task whose required production code change can be localized and validated.

Concept-propagation mining. We traverse non-merge commits from 2018 onward and identify candidate anchor commits that modify both Python production code and candidate tests while introducing at least one named definition. When several definitions are introduced, a root concept is selected by deterministic type priority (exception, class, dataclass, enumeration, protocol, then public function), with lexical tie-breaking. For each root concept, we search subsequent commits within a three-year window and retain candidates that modify both production code and tests. Propagation candidates are generated from newly added production lines using hybrid AST and lexical matching for exception use, inheritance, and call- or interface-related concept use. Because diff fragments are often syntactically incomplete, lexical matching also recovers otherwise unparseable candidates. Surviving evidence is subsequently filtered by explicit syntactic roles and manually reviewed for semantic validity.

Chain consolidation and semantic filtering. A propagation commit may itself introduce a concept that later propagates; in this case, we connect the resulting child chain to its parent and merge their commits into a single chronological execution chain. Duplicate commits and propagation events are removed, structural-only changes are filtered, and chains shorter than three unique commits are discarded. The remaining chains are manually reviewed end-to-end before model evaluation, with semantically trivial relations removed and the minimum chain length reapplied.

Executable task reconstruction. For each retained commit c_i, we reconstruct a date-compatible execution environment and identify its target oracle O_i. We retain only records for which the oracle passes on the complete target snapshot S_i, excluding runs with no executed tests or only skipped tests on WSL.

Let W_i denote the complete production-code change introduced by c_i, and let $\mathcal{F}$ denote conclusive non-passing outcomes. We localize the mined propagation evidence to its smallest enclosing function, asynchronous function, or class, obtaining implementation scopes Σ_i and the task-specific change $\Delta_i^{\mathrm{code}} = W_i|_{\Sigma_i}$. A retained task must satisfy

$$\mathrm{Oracle}_i(S_i) = \mathrm{PASS}, \quad \mathrm{Oracle}_i(S_i \ominus W_i) \in \mathcal{F}, \quad \text{and} \quad \mathrm{Oracle}_i(S_i \ominus \Delta_i^{\mathrm{code}}) \in \mathcal{F}. \tag{3}$$

For the scope-level ablation, selected definitions are restored to their parent versions or removed if target-only, while code outside Σ_i remains in the target state.

Behavioural task descriptions. For each validated task, we use GPT-5 as a separate task-description generator which receives the complete parent and target implementations of the selected scopes as private construction evidence and is prompted to produce an implementation-underspecified, issue-style task description τ_i that states the problem context and expected behaviour without revealing the target implementation or patch. Descriptions undergo iterative LLM-assisted auditing and manual adjudication for grounding, completeness, and implementation leakage; none is revised using agent outcomes.

3.4 BENCHMARK STATISTICS

The final DR4Code-Bench contains 22 concept-propagation chains and 88 executable tasks from 10 Python repositories; Appendix A reports the per-repository breakdown. Chain lengths range from 3 to 11 (mean 4.0, median 3) and are dominated by short chains: 14 chains contain 3 tasks, 4 contain 4 tasks, and one each contains 5, 6, 8, and 11 tasks. Of the 88 tasks, 22 are anchors and 66 are history-bearing: 22 have $h_i = 1$, 30 have $h_i = 2$–3, and 14 have $h_i \geq 4$, where h_i is the number of earlier tasks in the chain.

4 EVALUATION

We evaluate how reusing Investigation Reports affects coding correctness and investigation effort, and examine the stability of these effects across repeated executions.

4.1 EXPERIMENTAL SETUP

History conditions. Let R_j^B be the Investigation Report produced by the No-History baseline on task T_j, and let B_i contain all available R_j^B with $j < i$. The four conditions differ only in the report history H_i supplied during research:

• **No-History**: The agent receives no prior report during research ($H_i = \varnothing$), providing the baseline for measuring the effect of report reuse.

• **Full-History**: The agent receives every available report from earlier tasks in the same chain ($H_i = B_i$), representing unfiltered accumulation of investigation experience.

• **Recent-1**: The agent receives the most recent available baseline report from an earlier task in the same chain: $H_i = \{R_{j^\star}^B\}$, where $j^\star = \max\{j < i : R_j^B \in B_i\}$. If $B_i = \varnothing$, then $H_i = \varnothing$.

• **Top-1**: The agent receives the report in B_i most similar to the current task description τ_i. Similarity is measured by cosine similarity between all-MiniLM-L6-v2 embeddings (Sentence Transformers).

Only baseline reports enter B_i, so no reuse strategy affects the history seen by another. All conditions receive the same τ_i and X_i; anchors are run only under No-History. Appendix B provides retrieval and isolation details.

Models. We evaluate six models: GPT-5.1 (OpenAI, 2025), DeepSeek-V4-Flash (DeepSeek-AI et al., 2026), Qwen3.8-Flash (Alibaba Cloud, 2026), Claude-Sonnet-4.6 (Anthropic, 2026), Grok-4.3 (xAI, 2026), and Gemini-3.8-Flash (Google DeepMind, 2026). All models use the same two-phase agent, tools, task scopes, and report schema. Inference configurations are detailed in Appendix B.

Metrics. For task T_i under condition p, let $y_{i,p} \in \{0, 1\}$ equal one only when the candidate, restricted to the validated task scopes, passes the hidden oracle O_i, and zero otherwise. Using these binary outcomes, we evaluate each history condition using the following measures and analyses. Appendix B details the outcome-evaluation procedure.

• **Pass rate and paired effects**: Following executable repository-level coding benchmarks such as SWE-Bench (Jimenez et al., 2024), we report pass rate over the 66 history-bearing tasks. To expose task-level changes hidden by aggregate rates, we also count *rescues*, where a No-History failure passes under reuse, and *regressions*, where a No-History success fails under reuse.

• **History-depth effects**: We compare each reuse strategy with No-History at intermediate history depth ($h_i = 2$–3) and deep history ($h_i \geq 4$). Tasks with $h_i = 1$ are excluded from comparisons among reuse strategies because there is only one possible historical report to select.

• **Resource use**: We report episode-level token and tool-call usage. Uncertainty is estimated with a paired bootstrap that resamples whole chains; Appendices C.1 and C.2 provide the detailed procedure.

Table 2: Single-run pass rates, paired outcome changes, and average resource use over the 66 history-bearing tasks. The three outcome columns use different denominators: pass rates are over all 66 tasks, whereas rescue percentages are over the model's baseline failures and regression percentages over its baseline successes. Token and tool-call measurements are episode-level averages. Bold indicates the best fixed reuse strategy and the lowest resource use within each model; underlining indicates the second-best fixed reuse strategy and the second-lowest resource use within each model.

Model	Policy	Pass rate (%)	Rescues	Regressions	Input tokens	Output tokens	Research calls	Coding calls
GPT-5.1	No-History	48.5	–	–	**157.6k**	8.6k	11.27	2.73
	Full-History	**51.5**	6 (17.6%)	4 (12.5%)	176.0k	9.4k	10.50	**2.17**
	Recent-1	50.0	6 (17.6%)	5 (15.6%)	172.4k	9.3k	**10.08**	2.94
	Top-1	50.0	6 (17.6%)	5 (15.6%)	164.7k	**8.2k**	10.11	2.74
DeepSeek-V4-Flash	No-History	59.1	–	–	1529.0k	27.3k	30.14	3.38
	Full-History	**68.2**	11 (40.7%)	5 (12.8%)	**985.7k**	26.8k	**22.92**	3.27
	Recent-1	59.1	6 (22.2%)	6 (15.4%)	1139.5k	**24.6k**	24.65	**2.83**
	Top-1	60.6	8 (29.6%)	7 (17.9%)	1141.6k	25.9k	23.82	3.08
Qwen3.8-Flash	No-History	57.6	–	–	**976.5k**	**23.4k**	26.85	**3.42**
	Full-History	50.0	7 (25.0%)	12 (31.6%)	1160.2k	26.6k	27.48	3.59
	Recent-1	**54.5**	9 (32.1%)	11 (28.9%)	999.5k	26.4k	**25.70**	3.45
	Top-1	51.5	7 (25.0%)	11 (28.9%)	1133.6k	28.9k	27.44	3.86
Claude-Sonnet-4.6	No-History	62.1	–	–	875.4k	15.0k	27.35	2.83
	Full-History	**66.7**	7 (28.0%)	4 (9.8%)	1060.3k	16.8k	25.62	2.86
	Recent-1	65.2	6 (24.0%)	4 (9.8%)	**811.8k**	14.6k	**22.68**	2.74
	Top-1	**66.7**	8 (32.0%)	5 (12.2%)	859.0k	**14.0k**	23.62	**2.47**
Grok-4.3	No-History	56.1	–	–	174.1k	10.3k	8.45	2.95
	Full-History	56.1	6 (20.7%)	6 (16.2%)	**123.4k**	7.9k	**6.73**	**2.42**
	Recent-1	42.4	4 (13.8%)	13 (35.1%)	141.6k	**7.7k**	6.76	2.77
	Top-1	**57.6**	6 (20.7%)	5 (13.5%)	136.4k	8.6k	6.91	2.80
Gemini-3.8-Flash	No-History	66.7	–	–	1486.6k	40.9k	27.53	**3.52**
	Full-History	71.2	5 (22.7%)	2 (4.5%)	1519.6k	39.9k	25.79	3.62
	Recent-1	**72.7**	6 (27.3%)	2 (4.5%)	**1390.2k**	**38.2k**	**25.14**	**3.52**
	Top-1	71.2	6 (27.3%)	3 (6.8%)	1600.5k	42.6k	25.41	4.02

4.2 Results

4.2.1 Single-Run Landscape Across Six Models

In the frozen six-model evaluation, at least one fixed reuse policy attains a higher observed pass rate than No-History for five of the six models (Table 2), although the strongest policy varies across models. Qwen3.8-Flash is the exception, with all three reuse policies below its No-History baseline. The largest observed pass-rate gain is $+9.1$ percentage points for DeepSeek-V4-Flash under Full-History.

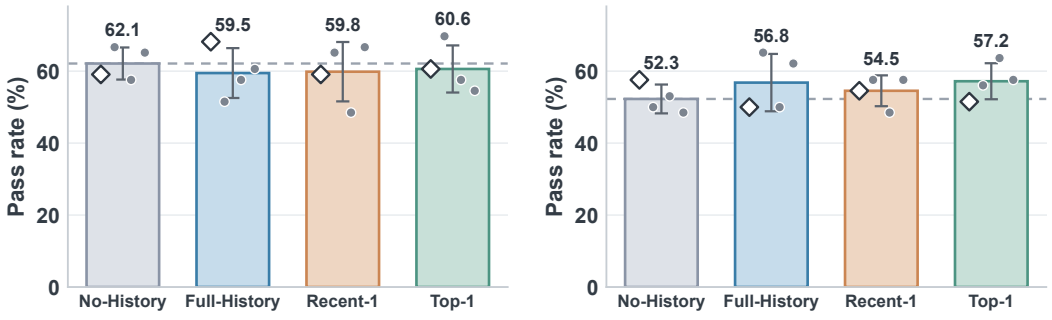
These aggregate differences conceal opposing task-level outcomes. All 18 model–policy comparisons contain both rescues of No-History failures and regressions of No-History successes. For example, DeepSeek-V4-Flash with Full-History rescues 11 tasks but regresses on five, yielding six additional passes overall. Thus, the observed gains do not represent uniformly better outcomes across tasks.

Taken at face value, these results suggest model- and policy-dependent correctness benefits from prior Investigation Reports. However, each task–condition pair is observed only once, so the apparent gains and strategy ordering may depend on the particular execution. We therefore next examine their stability under repeated runs.

4.2.2 Repeated-Run Reliability of Correctness Effects

To assess whether the pass-rate differences and policy rankings observed in the single-run evaluation persist across executions, we conduct three additional runs for DeepSeek-V4-Flash and Qwen3.8-Flash on all 66 history-bearing tasks under the same four history conditions. Together with the original evaluation, this yields four runs per condition. Anchor reports are fixed across runs, whereas non-anchor reports are regenerated from preceding No-History episodes within each run, so the experiment measures the stability of the end-to-end reuse protocol.

Repeated execution substantially changes the aggregate picture (Figure 2). For DeepSeek-V4-Flash, Full-History exceeds No-History by 9.1 percentage points in the original run, but its four-run mean falls below No-History (59.5% vs. 62.1%). Qwen3.8-Flash shows the opposite pattern: all three reuse policies underperform No-History in the original run, whereas their four-run means all exceed the No-History mean, with Top-1 highest at 57.2% versus 52.3%. Thus, both the direction of the observed reuse effect and the ordering of fixed reuse policies can change across executions. Run-to-run sign changes also persist when module-level imports are retained in the three additional runs (Appendix C.7), indicating that discarded imports alone do not explain this variability.



(a) DeepSeek-V4-Flash (b) Qwen3.8-Flash

Figure 2: Repeated-run pass rates on the 66 history-bearing tasks (four runs per condition). Bars: four-run mean; error bars: ± 1 SD across runs (run-to-run variability, not confidence intervals). Dashed line: No-History mean; diamond: Original run; circles: repeats.

4.2.3 Resource Efficiency

Research and coding interactions. Across the six models and 66 history-bearing tasks, Full-History, Recent-1, and Top-1 use 2.09, 2.77, and 2.38 fewer research calls per episode than No-History, respectively (Table 2). Mean coding-call differences are small (-0.15, -0.10, and $+0.02$), with all three 95% chain-bootstrap intervals including zero. Research-call counts here include successful tool executions only. Restricting comparisons to paired episodes in which research completes under both conditions yields negative mean differences in all 18 model–policy comparisons, with equally weighted mean reductions of 2.24, 2.93, and 2.67 calls, respectively.

Repeated-run evidence. Across four runs, research-call reductions average 5.14–5.56 calls for DeepSeek and 0.52–2.28 for Qwen. All three policy-specific 95% chain-bootstrap intervals exclude zero for DeepSeek, compared with only Recent-1 for Qwen. Restricting comparisons to pairs with recorded call counts and completed research preserves negative mean differences in all six model–policy comparisons. These restricted-sample results are descriptive checks conditional on research completion. An audit of complete research traces from the three additional runs shows that including failed tool invocations also preserves negative mean differences in all six model–policy comparisons (Appendix C.1).

Tokens per solved task. We divide total input and output tokens across all episodes, including failures, by the number of successful episodes, pooling runs for the repeated-run analysis. DeepSeek's Full-History reduction changes from 43.6% in the original run to 9.6% when pooling four runs. All six repeated-run model–policy estimates indicate reductions, but their 95% chain-bootstrap

Table 3: Pass-rate differences (pp) by history depth: policies vs. No-History; Δ = Full-History $-$ Recent-1. Models are equally weighted; counts denote distinct tasks. Four-run means include Original. All values are descriptive point estimates, while contrasts are computed before rounding.

History depth	Tasks	Full-History	Recent-1	Top-1	Δ
Six models (single run)					
$h_i = 2$–3	30	$+3.33$	$+2.22$	$+3.89$	$+1.11$
$h_i \geq 4$	14	$+2.38$	-7.14	-4.76	$+9.52$
DeepSeek + Qwen (four-run mean)					
$h_i = 2$–3	30	$+0.42$	$+3.33$	$+3.33$	-2.92
$h_i \geq 4$	14	$+3.57$	$+0.89$	$+1.79$	$+2.68$

intervals include zero. The metric definition and detailed token-efficiency results are provided in Appendix C.2.

4.2.4 History Characteristics and Reuse Outcomes

History depth. The intermediate-history subset contains 30 tasks, while the deep-history subset contains 14 tasks from four chains. In the six-model single-run evaluation, the Full-History advantage over Recent-1 increases from 1.11 percentage points at intermediate depth ($h_i = 2$–3) to 9.52 points at deep history ($h_i \geq 4$; Table 3). Leaving out each contributing deep-history chain in turn keeps this contrast positive at 8.33–10.26 points (Appendix C.3). Averaged equally across DeepSeek and Qwen and all four runs, the corresponding contrasts are -2.92 and 2.68 points; all three policies exceed No-History on average in the deep-history subset (Table 3).

Report provenance. We group reports by whether their source No-History episodes passed the oracle. For Full-History, passing and non-passing groups contain exclusively passing or non-passing sources; mixed histories are reported separately and empty histories excluded. Passing-source groups have higher downstream pass rates in all 18 single-run comparisons and five of six pooled four-run comparisons; Qwen with Full-History is the exception. However, these groups also have higher current-task No-History pass rates, while rescue-rate contrasts vary. Source success is therefore associated with downstream success, without establishing greater reuse benefit or report quality. Appendix C.4 provides detailed group counts and rescue rates.

Report selection. On the 44 tasks with history depth $h_i \geq 2$, Recent-1 and Top-1 supply different reports on 43.2–61.4% of tasks across the six models. For Qwen, five of these 44 tasks have fewer than two available reports. Similarity-based selection therefore does not simply reproduce recency-based selection. On the same subset, every model has tasks solved by Recent-1 but not Top-1, and vice versa. Each policy achieves the higher pass rate for three of the six models. The pass-rate differences (Top-1 minus Recent-1) range from -4.55 to $+15.91$ percentage points on this subset. Appendix C.5 provides the selection and outcome breakdowns.

5 Conclusion

We introduced DR4Code-Bench, an executable benchmark for evaluating pre-solution Investigation Report reuse across concept-linked software evolution tasks. Its independently reconstructed tasks and report-only cross-task transfer enable separate measurement of coding correctness and investigation effort. Across six models, reuse reduces research-tool calls on average, with reductions also observed when comparisons are restricted to episodes in which research completes under both conditions. Three additional runs for DeepSeek and Qwen show that observed correctness gains and policy rankings can change across executions, while research-call reductions are more consistent for DeepSeek than for Qwen. In the four-run analysis, all six model–policy estimates indicate fewer tokens per solved task than No-History, although their 95% chain-bootstrap intervals include zero. These findings distinguish reduced investigation-tool use from improved coding correctness and motivate evaluating both outcomes when assessing cross-task report reuse. DR4Code-Bench provides a controlled setting for developing report-selection methods that jointly consider investigation cost and downstream coding outcomes across evolving repository states.

AI USE STATEMENT

Generative AI tools were used to assist with behavioural task description generation and auditing, diagnosis of benchmark-environment compatibility issues, development of analysis code, interpretation of experimental results, and polishing passages to improve the clarity of this paper.

REFERENCES

Alibaba Cloud. Qwen3.8-Flash, August 2026. URL https://help.aliyun.com/en/model-studio/qwen3-8-flash. Model documentation.

Anthropic. Claude Sonnet 4.6 System Card, February 2026. URL https://anthropic.com/claude-sonnet-4-6-system-card. System Card.

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

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

Zhaoling Chen, Robert Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. LocAgent: Graph-guided LLM agents for code localization. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8697–8727, Vienna, Austria, July 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.acl-long.426.

DeepSeek-AI et al. DeepSeek-V4: Towards Highly Efficient Million-Token Context Intelligence. *arXiv preprint arXiv:2606.19348*, 2026. doi: 10.48550/arXiv.2606.19348. URL https://arxiv.org/abs/2606.19348.

Gangda Deng, Zhaoling Chen, Zhongming Yu, Haoyang Fan, Yuhong Liu, Yuxin Yang, Dhruv Parikh, Rajgopal Kannan, Le Cong, Mengdi Wang, Qian Zhang, Viktor Prasanna, Xiangru Tang, and Xingyao Wang. SWE-Milestone: Evaluating AI agents on continuous software evolution. *arXiv preprint arXiv:2603.13428*, 2026.

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

Google DeepMind. Gemini 3.8 Flash, September 2026. URL https://deepmind.google/models/model-cards/gemini-3-8-flash/. Model Card.

Zexue He, Yu Wang, Churan Zhi, Yuanzhe Hu, Tzu-Ping Chen, Lang Yin, Ze Chen, Tong Arthur Wu, Siru Ouyang, Zihan Wang, et al. MemoryArena: Benchmarking agent memory in interdependent multi-session agentic tasks. *arXiv preprint arXiv:2602.16313*, 2026.

Yuanzhe Hu, Yu Wang, and Julian McAuley. Evaluating memory in LLM agents via incremental multi-turn interactions. *arXiv preprint arXiv:2507.05257*, 2025.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=chfJJYC3iL.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *The Twelfth International Conference on Learning Representations*, 2024.

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

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

OpenAI. GPT-5.1 Instant and GPT-5.1 Thinking System Card Addendum, November 2025. URL https://openai.com/index/gpt-5-system-card-addendum-gpt-5-1/. System Card Addendum.

Sentence Transformers. all-MiniLM-L6-v2. URL https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2. Hugging Face Model Card.

Ramneet Singh, Sathvik Joel, Abhav Mehrotra, Nalin Wadhwa, Ramakrishna B. Bairi, Aditya Kanade, and Nagarajan Natarajan. Code Researcher: Deep research agent for large systems code and commit history. *arXiv preprint arXiv:2506.11060*, 2025.

Haoran Tan, Zeyu Zhang, Chen Ma, Xu Chen, Quanyu Dai, and Zhenhua Dong. MemBench: Towards more comprehensive evaluation on the memory of LLM-based agents. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 19336–19352, Vienna, Austria, July 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.findings-acl.989.

Yuyao Wang, Zhongjian Zhang, Mo Chi, Kaichi Yu, Yuhan Li, Miao Peng, Bing Tong, Chen Zhang, Yan Zhou, and Jia Li. EvoMemBench: Benchmarking agent memory from a self-evolving perspective. *arXiv preprint arXiv:2605.18421*, 2026.

Di Wu, Hongwei Wang, Wenhao Yu, Yuwei Zhang, Kai-Wei Chang, and Dong Yu. LongMemEval: Benchmarking chat assistants on long-term interactive memory. In *The Thirteenth International Conference on Learning Representations*, 2025.

xAI. Grok 4.3, 2026. URL https://docs.x.ai/developers/models/grok-4.3. Model documentation.

Shaoqiu Zhang, Yuhang Wang, Jialiang Liang, Yuling Shi, Wenhao Zeng, Maoquan Wang, Shilin He, Ningyuan Xu, Siyu Ye, Kai Cai, and Xiaodong Gu. SWE-Explore: Benchmarking how coding agents explore repositories. *arXiv preprint arXiv:2606.07297*, 2026.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, pp. 1592–1604, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706127. doi: 10.1145/3650212.3680384.

Jiayuan Zhu, Junde Wu, Minhao Hu, Shengda Zhu, Jiazhen Pan, Weixiang Shen, Yijun Yang, Fenglin Liu, Jianye Hao, Yueming Jin, Qirong Ho, and Min Xu. SWE Context Bench: A benchmark for context learning in coding. *arXiv preprint arXiv:2602.08316*, 2026.

A Benchmark Construction and Composition

A.1 Repository Composition

DR4Code-Bench contains 88 executable tasks organized into 22 concept-propagation chains across ten source repositories. Each chain begins with one anchor task, followed by tasks eligible to reuse investigation reports from earlier tasks in the same chain. Table 4 summarizes the repository-level distribution. The 22 anchors are excluded from reuse-effect analyses, leaving 66 history-bearing tasks. Here, history-bearing denotes the availability of preceding tasks in a chain; the reports actually available to an agent depend on the outcomes of the preceding research episodes.

Table 4: Repository-level composition of DR4Code-Bench. Each chain contains one anchor task. Reuse tasks are the remaining, non-anchor tasks that form the 66-task subset used to evaluate investigation-memory reuse.

Repository	**Chains**	**Tasks**	**Reuse tasks**
Hypothesis/h	7	24	17
pip	4	13	9
HTTPX	3	11	8
Rich	1	11	10
Django	2	9	7
AnyIO	1	8	7
Celery	1	3	2
Pydantic	1	3	2
Requests	1	3	2
SQLAlchemy	1	3	2
Total	**22**	**88**	**66**

A.2 Chain-Level Concept Labels

Table 5 lists the concept labels associated with the 22 retained chains and the number of tasks in each chain, including its anchor. These labels summarize the concepts linking historical changes; they are not exhaustive specifications of the function or class scopes reconstructed for individual tasks.

Where multiple labels are shown, arrows indicate their order along the chain, rather than necessarily denoting symbol renaming, inheritance, or a direct call relation. Parenthetical annotations clarify the intended meaning of otherwise broad labels.

Table 5: Concept labels for the 22 retained benchmark chains. Task counts include anchor tasks. Chain identifiers follow the dataset's batch-local numbering.

Repository / chain	Tasks	Concept label(s)
`httpx / 0`	4	`InvalidURL`
`httpx / 1`	3	`merge_url`
`httpx / 2`	4	`RedirectError` → `RedirectBodyUnavailable` → `NotRedirectResponse`
`pydantic / 3`	3	`make_generic_field_serializer`
`requests / 4`	3	`InvalidJSONError` → `JSONDecodeError`
`sqlalchemy / 5`	3	`UnevaluatableError`
`anyio / 0`	8	`shield` → `CancelScope`
`rich / 1`	11	`plain` → `align`
`celery / 0`	3	`BeatLazyFunc`
`django / 1`	6	`ColPairs`
`django / 2`	3	`MediaAsset`

Continued on next page

Table 5 continued

Repository / chain	Tasks	Concept label(s)
h / 3	5	groupid (validation/uniqueness)
h / 4	4	client_authority
h / 5	3	Organization
h / 6	3	is_authenticated
h / 7	3	ConflictError
h / 8	3	Priority
h / 9	3	profile_groups
pip / 10	4	location (distribution/editable-project location)
pip / 11	3	proxy (propagation to isolated builds)
pip / 12	3	raw_name
pip / 13	3	make_editable_error

A.3 Maintenance-Noise Filtering

Table 6 lists the commit-message substrings used by the maintenance-noise filter during candidate collection.

Table 6: Commit-message substrings used by the maintenance-noise filter.

Category	Lowercase substrings
Merge and release	merge, merge pull request, merge branch, release, release v, version, bump version, upgrade version, bump
Documentation	changelog, typo, docs only, documentation, readme, doc
Formatting and tooling	lint, formatting, ci , pre-commit, mypy, ruff, black, isort, flake8, pylint
Deprecation and cleanup	drop support for, remove deprecated, deprecate, deprecated, cleanup unneeded

B Additional Experimental Details

Report Selection. Recent-1 selects the most recent available baseline report from an earlier task in the same chain, which need not come from the immediately preceding task if that task produced no report. For Top-1, each preceding baseline report is represented by concatenating its task-relevant summary, concept descriptions, and relevant-file information. Top-1 selects the report whose all-MiniLM-L6-v2 embedding has the highest cosine similarity to the embedding of the current task description τ_i. When the report bank is empty, all three reuse conditions receive an empty history.

Episode isolation. The repository is reset between conditions, and candidate code from one episode is never visible to another. Reuse conditions never contribute reports to the historical bank. The anchor task T_1 has no preceding report and is therefore executed only under No-History. At $h_i = 1$, all three reuse policies receive identical historical input: the anchor report if available, or an empty history otherwise. Differences among these policies at this depth therefore do not reflect report selection.

Inference configuration. We do not explicitly set temperature, sampling seed, or reasoning effort, leaving these settings to the provider endpoints. DeepSeek-V4-Flash refers to the 0731 version throughout. For Claude, the maximum output length is set to 65,536 tokens per response. Results are reported separately for each model.

Episode termination. The agent loops impose no explicit stage-level limit on interaction rounds or tool calls. Research completes when the agent returns a report that passes JSON parsing and the

required structural checks; invalid reports trigger a request for corrected output. Coding completes when the final non-empty response line is `DONE`, after which the candidate is evaluated.

Outcome evaluation. After an episode finishes, the evaluator retains only candidate implementations within the validated task scopes, restores the target repository state and hidden oracle O_i, and reapplies the scoped candidate modification. Candidate edits outside the validated scopes, modifications to tests, and agent-created files outside the target state are discarded. We set $y_{i,p} = 1$ only when the resulting scoped candidate passes the target oracle; all other agent-originated outcomes are counted as failures.

C Additional Experimental Results

C.1 Research-Call Reductions and Completion Sensitivity

Metric and comparison samples. Research-call counts record successful research-tool executions; calls flagged as tool errors are excluded by the counter. A successful execution need not return useful information; an empty search result also counts. For each model and policy, we subtract the No-History research-call count from the reuse count for the same task, then average these paired differences. In repeated evaluations, each pair also belongs to the same run. Differences are defined as reuse minus No-History, so negative values indicate fewer recorded calls per episode. The full sample includes all evaluated episodes, including those whose research stage ends with an error or whose final coding outcome is unsuccessful. An error does not automatically set the count to zero: successful tool executions recorded before termination remain counted.

We additionally restrict comparisons to pairs with recorded call counts and completed research under both conditions. Completion means that the research stage returns an accepted report, independently of the final coding outcome. It does not require a positive research-call count or verify the quality of the report. If either condition lacks a recorded count or completed research, the entire pair is excluded from this restricted comparison. Consequently, retained sample sizes can differ across models and policies.

For the six-model summary, we average model-specific mean differences with equal model weights. For the four-run analysis, we pool task–run pairs across the original run and three additional runs. In the completion-restricted sample, runs therefore contribute in proportion to their retained pair counts.

Uncertainty estimation. We use 10,000 paired whole-chain bootstrap resamples. Each resample draws 22 chains with replacement and retains the paired observations within each sampled chain. For the completion-restricted analysis, each resample's mean is computed only from pairs with recorded call counts and completed research under both conditions. Within each bootstrap resample, we use the same sampled chains for all six models and average their model-specific mean differences with equal weights. For repeated-run analyses, all observed runs included in the analysis for each sampled task are retained together; runs are also not resampled independently. This means four runs for the primary repeated-run analysis and three additional runs for the attempted-call audit below. We report marginal 95% percentile intervals, using the 2.5th and 97.5th percentiles, without multiplicity adjustment. Repeated-run intervals are conditional on the observed runs, rather than prediction intervals for future executions. All bootstrap analyses use fixed random seeds for reproducibility.

Six-model single-run results. Table 7 compares the full sample with the completion-restricted sample. All 18 model–policy point estimates are negative after restricting to paired completed-research episodes. The equally weighted mean differences are -2.24, -2.93, and -2.67 calls for Full-History, Recent-1, and Top-1, respectively, with marginal 95% intervals of $[-3.37, -1.27]$, $[-4.42, -1.81]$, and $[-4.11, -1.61]$. These intervals describe the aggregate estimates, not the significance of every model-specific difference.

Pooled repeated-run results. Table 8 reports comparisons pooled across four runs. In the full sample, all three DeepSeek intervals lie below zero, whereas only Recent-1 does so for Qwen. After restricting to paired completed-research episodes, all six point estimates remain negative. Within this subset, intervals lie below zero for all three DeepSeek policies and for Qwen's Full-History and Recent-1; Qwen's Top-1 interval includes zero.

Table 7: Mean successful research-tool-call differences from No-History in the six-model single-run evaluation, in calls per episode. All uses 66 task pairs per model–policy comparison. Completed retains pairs with recorded counts and completed research under both conditions; n_{comp} gives the retained pair count. Negative values indicate fewer calls. The final three rows report equally weighted model means and summed pair counts across models.

Model	Policy	All	Completed	n_{comp}
GPT-5.1	Full-History	-0.77	-0.77	66
	Recent-1	-1.20	-1.20	66
	Top-1	-1.17	-1.17	66
DeepSeek-V4-Flash	Full-History	-7.21	-6.98	65
	Recent-1	-5.48	-4.64	64
	Top-1	-6.32	-6.20	65
Qwen3.8-Flash	Full-History	$+0.64$	-0.47	55
	Recent-1	-1.15	-3.00	55
	Top-1	$+0.59$	-1.29	56
Claude-Sonnet-4.6	Full-History	-1.73	-1.73	66
	Recent-1	-4.67	-4.67	66
	Top-1	-3.73	-3.73	66
Grok-4.3	Full-History	-1.73	-1.73	66
	Recent-1	-1.70	-1.70	66
	Top-1	-1.55	-1.55	66
Gemini-3.8-Flash	Full-History	-1.74	-1.74	66
	Recent-1	-2.39	-2.39	66
	Top-1	-2.12	-2.12	66
Equal-model mean	Full-History	-2.09	-2.24	384
	Recent-1	-2.77	-2.93	383
	Top-1	-2.38	-2.67	385

Table 8: Mean successful research-tool-call differences from No-History pooled across four runs, in calls per episode. All pairs contains 264 task–run pairs per model–policy comparison. Completed requires recorded counts and completed research under both conditions. n counts retained task–run pairs, not distinct tasks. Intervals are marginal 95% paired whole-chain bootstrap intervals. Negative differences indicate fewer calls.

	All pairs		Completed		
Policy	Difference	95% CI	n	Difference	95% CI
DeepSeek-V4-Flash					
Full-History	-5.56	$[-8.11, -3.60]$	251	-5.84	$[-8.36, -3.91]$
Recent-1	-5.22	$[-8.02, -3.14]$	255	-5.38	$[-8.07, -3.41]$
Top-1	-5.14	$[-7.41, -3.33]$	254	-5.22	$[-7.42, -3.53]$
Qwen3.8-Flash					
Full-History	-2.09	$[-4.24, +0.20]$	220	-3.00	$[-5.18, -0.98]$
Recent-1	-2.28	$[-4.03, -0.85]$	215	-3.07	$[-5.35, -1.35]$
Top-1	-0.52	$[-2.50, +1.25]$	225	-1.84	$[-4.00, +0.01]$

Zero-count sensitivity. Recorded zeros are retained in the analyses above. An accepted report can be returned without a successful research-tool execution, and the subsequent coding stage can perform its own repository inspection. A zero recorded research-call count therefore does not imply that research failed or that no tool calls were attempted.

As a separate sensitivity check, we exclude pairs with a zero recorded research-call count in either condition. We recompute means and 10,000 paired whole-chain bootstrap intervals using only retained pairs within each sampled chain, without imposing an additional research-completion restriction. Table 9 reports the results. The six-model equally weighted mean differences remain negative, at -2.05, -2.58, and -2.38 calls for Full-History, Recent-1, and Top-1, respectively, with all three intervals below zero. All six four-run mean differences also remain negative; all three DeepSeek intervals and Qwen's Recent-1 interval lie below zero, whereas Qwen's Full-History and Top-1 intervals include zero. Thus, whether these intervals include zero is unchanged from the corresponding full-sample analyses. Individual estimates can nevertheless be sensitive: Qwen's original-run Recent-1 difference changes from -1.15 to -0.05 calls after excluding two zero-count pairs.

Table 9: Successful research-tool-call differences after excluding pairs with a zero count in either condition. Differences are reuse minus No-History, in calls per episode. Intervals are marginal 95% percentile intervals from 10,000 paired whole-chain bootstrap resamples. The six-model summary averages model-specific means equally; its n is the total retained task–model pair count. For each repeated model, n counts retained task–run pairs across the original run and three additional runs, retained together during resampling. No additional research-completion restriction is applied.

Policy	n	Difference	95% CI
Six-model equal-weight summary (single run)			
Full-History	393	-2.05	$[-3.22, -0.97]$
Recent-1	394	-2.58	$[-4.17, -1.24]$
Top-1	396	-2.38	$[-3.79, -1.18]$
DeepSeek-V4-Flash (four runs)			
Full-History	263	-5.58	$[-8.14, -3.61]$
Recent-1	262	-5.17	$[-8.01, -3.10]$
Top-1	263	-5.16	$[-7.45, -3.33]$
Qwen3.8-Flash (four runs)			
Full-History	264	-2.09	$[-4.24, +0.20]$
Recent-1	262	-2.02	$[-3.68, -0.70]$
Top-1	263	-0.40	$[-2.29, +1.35]$

Sensitivity to failed-call accounting. For the three additional runs of DeepSeek and Qwen, we also count attempted research-tool calls, including calls that return tool errors or terminate research with an exception. Table 10 compares the two counting definitions on the same observations. All six pooled mean differences become more negative when failed invocations are included. Five attempted-call intervals exclude zero, while Qwen's Top-1 interval includes zero. Restricting to pairs with completed research under both conditions also preserves negative attempted-call means in all six comparisons. The direction of the reductions therefore persists under both counting definitions in these additional runs.

Interpretation. The aggregate direction of the observed research-call reductions persists after restricting to completed research or excluding zero-count pairs. The three-additional-run audit also shows that this direction persists when failed tool invocations are counted. Fewer calls do not by themselves establish equal investigation quality, lower token expenditure, or improved coding correctness.

C.2 Token Efficiency

Metric definition. For each model and policy p, we compute tokens per solved task as

$$\mathrm{TPS}_p = \frac{\sum_{r=1}^{R} \sum_{i=1}^{66} \left(t_{i,p,r}^{\mathrm{in}} + t_{i,p,r}^{\mathrm{out}} \right)}{\sum_{r=1}^{R} \sum_{i=1}^{66} y_{i,p,r}}, \tag{4}$$

Table 10: Mean research-tool-call differences from No-History under two counting definitions, pooled across the three additional runs only. Each model–policy comparison contains 198 task–run pairs (66 tasks $\times$ 3 runs). Successful excludes failed invocations; Attempted includes them but excludes requested calls that were never executed. Differences and interval endpoints are in calls per episode; negative values indicate fewer calls. Attempted-call intervals are marginal 95% intervals from 10,000 paired whole-chain bootstrap resamples, retaining all three runs together, without multiplicity adjustment.

Policy	Successful	Attempted	Attempted 95% CI
DeepSeek-V4-Flash			
Full-History	-5.01	-5.29	$[-8.36, -2.80]$
Recent-1	-5.14	-5.72	$[-8.92, -3.30]$
Top-1	-4.74	-5.13	$[-7.67, -3.06]$
Qwen3.8-Flash			
Full-History	-3.00	-3.41	$[-5.85, -1.09]$
Recent-1	-2.66	-2.97	$[-5.01, -1.37]$
Top-1	-0.89	-1.23	$[-3.50, +0.84]$

where $t^{\text{in}}_{i,p,r}$ and $t^{\text{out}}_{i,p,r}$ denote episode input and output token counts, and $y_{i,p,r} = 1$ only if the episode passes the target oracle. We use $R = 1$ for the single-run evaluation and $R = 4$ for the pooled repeated-run analysis. Failed episodes contribute to the numerator. Successes on the same task in different runs are counted separately in the denominator. Thus, the repeated-run metric pools token consumption and success counts rather than averaging run-specific ratios.

Token counts cover the current episode's research and coding stages, including historical reports supplied as input. The cost of generating those reports in earlier episodes is not additionally charged to the reuse condition.

Relative changes from the same model's No-History baseline are reported as percentages:

$$\Delta_p^{\text{TPS}} = 100 \left(\frac{\text{TPS}_p}{\text{TPS}_{\text{No-History}}} - 1 \right). \tag{5}$$

Negative values indicate fewer tokens per successful episode.

Uncertainty estimation. We use 10,000 paired whole-chain bootstrap resamples. Each resample draws 22 chains with replacement, retaining all history-bearing tasks within each sampled chain and preserving the pairing between each reuse policy and No-History. Tokens per solved task and their relative changes are recomputed within each resample. Marginal 95% intervals use the 2.5th and 97.5th percentiles.

For the six-model single-run analysis, we additionally report Bonferroni-adjusted intervals across the 18 model–policy comparisons, using percentile levels $100(0.025/18)$ and $100(1 - 0.025/18)$. These intervals target 95% family-wise coverage. For the repeated-run analysis, all four observed runs of each sampled task are retained together; runs are not resampled independently. The intervals are conditional on the four observed runs, rather than prediction intervals for future executions.

Six-model single-run results. Table 11 reports absolute tokens per solved task, while Figure 3 shows relative changes and their uncertainty.

Pooled repeated-run results. Figure 4 reports relative changes for DeepSeek and Qwen after pooling the original run and three additional runs. Each model–policy condition contains 264 episodes (66 tasks $\times$ 4 runs), evaluated with the same scoped outcome evaluator. All six point estimates indicate reductions, but all six marginal 95% intervals include zero.

Table 11: Tokens per solved task over the 66 history-bearing tasks, in millions. Each value is the total input and output tokens consumed by all episodes under a model–policy condition, including failures, divided by the number of solved tasks. Values are descriptive point estimates.

Model	No-History	Full-History	Recent-1	Top-1
GPT-5.1	0.343	0.360	0.363	0.346
DeepSeek-V4-Flash	2.634	1.485	1.970	1.926
Qwen3.8-Flash	1.737	2.374	1.881	2.257
Claude-Sonnet-4.6	1.433	1.616	1.268	1.310
Grok-4.3	0.329	0.234	0.352	0.252
Gemini-3.8-Flash	2.291	2.190	1.964	2.307

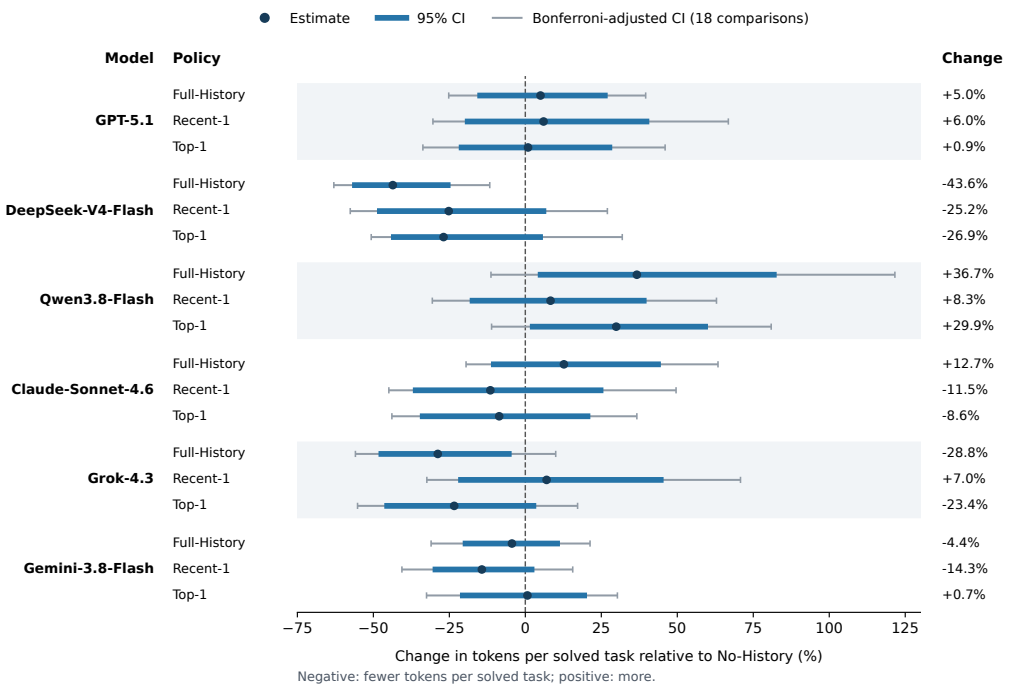


Figure 3: Relative changes in tokens per solved task compared with the same model's No-History baseline. Points indicate estimates, thick blue segments show marginal 95% percentile bootstrap confidence intervals, and thin grey segments show Bonferroni-adjusted intervals across the 18 model–policy comparisons, targeting 95% family-wise coverage. All estimates and interval endpoints are percentages; negative values indicate reductions. The dashed vertical line denotes no change.

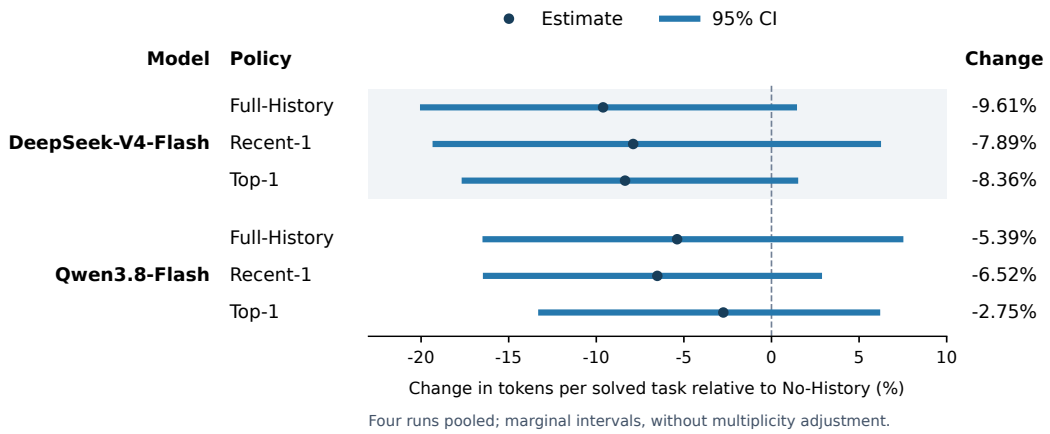


Figure 4: Relative changes in tokens per solved task after pooling four runs over the 66 history-bearing tasks, compared with each model's pooled No-History baseline. Points show estimates; blue segments show marginal 95% intervals from 10,000 paired whole-chain bootstrap resamples, without multiplicity adjustment. Negative values indicate reductions. The dashed vertical line denotes no change.

C.3 History-Depth Analysis

Table 12 provides the per-model breakdown underlying the aggregate depth comparison in the main text. Table 13 assesses whether the deep-history contrast is sensitive to any single chain. The leave-one-chain-out results are descriptive sensitivity checks, not independent repeated experiments.

Table 12: Pass-rate differences from each model's No-History baseline by history depth, in percentage points. The intermediate-depth group ($h_i = 2$–3) contains 30 tasks; the deep-history group ($h_i \geq 4$) contains 14 tasks from four chains.

	Effect of history depth (Δpp)					
	$h = 2$–3			$h \geq 4$		
Model	**Full-History**	**Recent-1**	**Top-1**	**Full-History**	**Recent-1**	**Top-1**
GPT-5.1	+10.0	+6.7	+10.0	0.0	−14.3	−28.6
DeepSeek-V4-Flash	+10.0	+6.7	+3.3	+7.1	−7.1	+14.3
Qwen3.8-Flash	0.0	0.0	−6.7	−14.3	−21.4	−21.4
Claude-Sonnet-4.6	+3.3	+6.7	+6.7	+14.3	0.0	+7.1
Grok-4.3	−3.3	−16.7	−3.3	+7.1	−14.3	+7.1
Gemini-3.8-Flash	0.0	+10.0	+13.3	0.0	+14.3	−7.1

Table 13: Leave-one-chain-out sensitivity analysis for the deep-history subset ($h_i \geq 4$), averaged over the six models. The policy columns report pass-rate differences from No-History; the final column reports the Full-History–Recent-1 difference. All differences are in percentage points. Each row recomputes outcomes over the remaining tasks, rather than evaluating predictions on the omitted chain.

Omitted chain	Tasks	Full-History	Recent-1	Top-1	Full-History − Recent-1
None	14	+2.38	−7.14	−4.76	+9.52
AnyIO chain 0	10	+3.33	−6.67	+1.67	+10.00
Rich chain 1	7	+4.76	−4.76	−9.52	+9.52
Django chain 1	12	0.00	−8.33	−6.94	+8.33
h chain 3	13	+2.56	−7.69	−5.13	+10.26

C.4 Report Provenance and Reuse Outcomes

Table 14 reports current-task outcomes by historical-report source. Table 15 restricts the analysis to tasks that No-History fails. These tables report descriptive sample outcomes with explicit group denominators. The six-model tables report the frozen single-run scoped evaluation. Table 16 additionally reports pooled outcomes across the original run and three additional runs for DeepSeek and Qwen, using the same scoped evaluator. Source labels are matched to the historical reports actually supplied in each run.

Table 14: Current-task outcomes by historical-report source. Cells show passed tasks over group size and the corresponding percentage. No-History and reuse are evaluated on the same tasks within each source group. For Full-History, the endpoint groups contain only non-passing or only passing source episodes; mixed histories are reported separately. Empty histories are excluded here: for Qwen's three empty-history tasks, No-History passes 2/3 and Full-History, Recent-1, and Top-1 pass 1/3, 2/3, and 0/3. Source labels describe earlier baseline episodes, not report correctness.

		Non-passing sources		Mixed sources		Passing sources	
Model	Policy	No-History	Reuse	No-History	Reuse	No-History	Reuse
GPT-5.1	Full-History	11/30 (36.7%)	12/30 (40.0%)	13/25 (52.0%)	15/25 (60.0%)	8/11 (72.7%)	7/11 (63.6%)
	Recent-1	16/41 (39.0%)	14/41 (34.1%)	–	–	16/25 (64.0%)	19/25 (76.0%)
	Top-1	16/42 (38.1%)	17/42 (40.5%)	–	–	16/24 (66.7%)	16/24 (66.7%)
DeepSeek-V4-Flash	Full-History	13/29 (44.8%)	17/29 (58.6%)	16/24 (66.7%)	17/24 (70.8%)	10/13 (76.9%)	11/13 (84.6%)
	Recent-1	18/34 (52.9%)	19/34 (55.9%)	–	–	21/32 (65.6%)	20/32 (62.5%)
	Top-1	22/41 (53.7%)	22/41 (53.7%)	–	–	17/25 (68.0%)	18/25 (72.0%)
Qwen3.8-Flash	Full-History	12/23 (52.2%)	9/23 (39.1%)	17/29 (58.6%)	17/29 (58.6%)	7/11 (63.6%)	6/11 (54.5%)
	Recent-1	14/32 (43.8%)	16/32 (50.0%)	–	–	22/31 (71.0%)	18/31 (58.1%)
	Top-1	20/39 (51.3%)	21/39 (53.8%)	–	–	16/24 (66.7%)	13/24 (54.2%)
Claude-Sonnet-4.6	Full-History	10/17 (58.8%)	8/17 (47.1%)	18/29 (62.1%)	19/29 (65.5%)	13/20 (65.0%)	17/20 (85.0%)
	Recent-1	17/28 (60.7%)	16/28 (57.1%)	–	–	24/38 (63.2%)	27/38 (71.1%)
	Top-1	13/24 (54.2%)	13/24 (54.2%)	–	–	28/42 (66.7%)	31/42 (73.8%)
Grok-4.3	Full-History	15/29 (51.7%)	16/29 (55.2%)	15/28 (53.6%)	14/28 (50.0%)	7/9 (77.8%)	7/9 (77.8%)
	Recent-1	18/37 (48.6%)	15/37 (40.5%)	–	–	19/29 (65.5%)	13/29 (44.8%)
	Top-1	21/43 (48.8%)	22/43 (51.2%)	–	–	16/23 (69.6%)	16/23 (69.6%)
Gemini-3.8-Flash	Full-History	10/19 (52.6%)	14/19 (73.7%)	18/27 (66.7%)	17/27 (63.0%)	16/20 (80.0%)	16/20 (80.0%)
	Recent-1	14/28 (50.0%)	16/28 (57.1%)	–	–	30/38 (78.9%)	32/38 (84.2%)
	Top-1	19/31 (61.3%)	22/31 (71.0%)	–	–	25/35 (71.4%)	25/35 (71.4%)

In the pooled four-run analysis, each model–policy condition contains 264 task–run observations before excluding empty histories. DeepSeek has no empty-history observations. Qwen has 12 under each reuse policy; on these observations, No-History passes 6/12, while Full-History, Recent-1, and Top-1 pass 7/12, 5/12, and 4/12, respectively. Their corresponding rescue counts are 3/6, 2/6, and 2/6.

Table 15: Rescue rates conditional on current-task No-History failure. Each cell reports rescued tasks over baseline-failed tasks in the indicated source group, followed by the rescue rate. Entries are descriptive sample proportions. Mixed Full-History sources are retained. Qwen has one baseline-failed task with empty history, excluded from these groups: its rescue outcomes are 0/1, 1/1, and 0/1 under Full-History, Recent-1, and Top-1. Dashes indicate inapplicable groups, not zero rescue rates.

Model	Policy	Non-passing sources	Mixed sources	Passing sources
GPT-5.1	Full-History	2/19 (10.5%)	4/12 (33.3%)	0/3 (0.0%)
	Recent-1	2/25 (8.0%)	–	4/9 (44.4%)
	Top-1	5/26 (19.2%)	–	1/8 (12.5%)
DeepSeek-V4-Flash	Full-History	7/16 (43.8%)	2/8 (25.0%)	2/3 (66.7%)
	Recent-1	4/16 (25.0%)	–	2/11 (18.2%)
	Top-1	4/19 (21.1%)	–	4/8 (50.0%)
Qwen3.8-Flash	Full-History	1/11 (9.1%)	5/12 (41.7%)	1/4 (25.0%)
	Recent-1	6/18 (33.3%)	–	2/9 (22.2%)
	Top-1	5/19 (26.3%)	–	2/8 (25.0%)
Claude-Sonnet-4.6	Full-History	0/7 (0.0%)	3/11 (27.3%)	4/7 (57.1%)
	Recent-1	2/11 (18.2%)	–	4/14 (28.6%)
	Top-1	3/11 (27.3%)	–	5/14 (35.7%)
Grok-4.3	Full-History	2/14 (14.3%)	3/13 (23.1%)	1/2 (50.0%)
	Recent-1	3/19 (15.8%)	–	1/10 (10.0%)
	Top-1	4/22 (18.2%)	–	2/7 (28.6%)
Gemini-3.8-Flash	Full-History	4/9 (44.4%)	1/9 (11.1%)	0/4 (0.0%)
	Recent-1	4/14 (28.6%)	–	2/8 (25.0%)
	Top-1	5/12 (41.7%)	–	1/10 (10.0%)

Table 16: Report provenance and current-task outcomes pooled across four runs. No-History and Reuse cells show successful episodes over the number of task–run observations in each source group, followed by percentages. Rescue cells show successful reuse episodes among the same group's No-History failures. For Full-History, Passing and Non-passing denote histories with exclusively passing or exclusively non-passing sources; mixed histories are reported separately. Counts are repeated observations, not distinct tasks. Empty histories are excluded from this table. All values are descriptive.

Model	Policy	Sources	No-History	Reuse	Rescue
DeepSeek-V4-Flash	Full-History	Non-passing	52/99 (52.5%)	50/99 (50.5%)	12/47 (25.5%)
		Mixed	74/115 (64.3%)	71/115 (61.7%)	10/41 (24.4%)
		Passing	38/50 (76.0%)	36/50 (72.0%)	7/12 (58.3%)
	Recent-1	Non-passing	71/124 (57.3%)	63/124 (50.8%)	9/53 (17.0%)
		Passing	93/140 (66.4%)	95/140 (67.9%)	15/47 (31.9%)
	Top-1	Non-passing	93/160 (58.1%)	88/160 (55.0%)	17/67 (25.4%)
		Passing	71/104 (68.3%)	72/104 (69.2%)	15/33 (45.5%)
Qwen3.8-Flash	Full-History	Non-passing	51/101 (50.5%)	55/101 (54.5%)	14/50 (28.0%)
		Mixed	52/105 (49.5%)	65/105 (61.9%)	23/53 (43.4%)
		Passing	29/46 (63.0%)	23/46 (50.0%)	3/17 (17.6%)
	Recent-1	Non-passing	60/132 (45.5%)	69/132 (52.3%)	21/72 (29.2%)
		Passing	72/120 (60.0%)	70/120 (58.3%)	16/48 (33.3%)
	Top-1	Non-passing	71/145 (49.0%)	80/145 (55.2%)	26/74 (35.1%)
		Passing	61/107 (57.0%)	67/107 (62.6%)	20/46 (43.5%)

C.5 RECENT-1 AND TOP-1 SELECTION

Table 17 compares Recent-1 and Top-1 on the 44 tasks at history depth $h_i \geq 2$. Selected-report differences are determined from the complete report contents recorded as inputs, rather than from episode outcomes. The depth criterion denotes chain position, not the number of successfully generated historical reports.

Table 17: Recent-1 versus Top-1 over the same 44 tasks at $h_i \geq 2$ for each model. Different report gives the number and percentage of tasks receiving different selected reports. The four outcome columns partition all 44 tasks, not only those with different selections. Fail denotes any non-PASS episode outcome. Counts are descriptive; outcome differences between independently executed episodes cannot be attributed solely to report selection.

Model	Different report	Both pass	Recent-1 only	Top-1 only	Both fail
GPT-5.1	25/44 (56.8%)	18	4	3	19
DeepSeek-V4-Flash	22/44 (50.0%)	23	4	6	11
Qwen3.8-Flash	19/44 (43.2%)	11	11	9	13
Claude-Sonnet-4.6	22/44 (50.0%)	26	3	4	11
Grok-4.3	23/44 (52.3%)	17	2	9	16
Gemini-3.8-Flash	27/44 (61.4%)	32	3	1	8

Recent-1 and Top-1 select different reports on 43.2–61.4% of this subset across models, showing that similarity-based selection does not simply reproduce recency-based selection. For Qwen3.8-Flash, 5 of the 44 tasks have fewer than two available reports; these tasks remain in the fixed denominator.

C.6 REPEATED-RUN HISTORY-DEPTH EFFECTS

Table 18 reports history-depth effects separately for DeepSeek-V4-Flash and Qwen3.8-Flash across the original evaluation and three additional runs. Each entry is the pass-rate difference, in percentage points, between a reuse strategy and the same model's No-History condition within the same run. The intermediate-depth subset contains 30 tasks, while the deep-history subset contains 14 tasks from four chains.

Table 18: Repeated-run pass-rate differences from the same run's No-History baseline, in percentage points, by history depth. The intermediate-depth subset contains 30 tasks; the deep-history subset contains 14 tasks from four chains.

Model	Run	Full-History	Recent-1	Top-1
Intermediate history: $h = 2$–3 (30 tasks)				
DeepSeek-V4-Flash	Original	+10.00	+6.67	+3.33
	Repeat 1	−13.33	+10.00	+3.33
	Repeat 2	+6.67	0.00	+10.00
	Repeat 3	+3.33	+10.00	−3.33
Qwen3.8-Flash	Original	0.00	0.00	−6.67
	Repeat 1	+10.00	+6.67	+3.33
	Repeat 2	−13.33	−10.00	+13.33
	Repeat 3	0.00	+3.33	+3.33
Deep history: $h \geq 4$ (14 tasks, 4 chains)				
DeepSeek-V4-Flash	Original	+7.14	−7.14	+14.29
	Repeat 1	−14.29	−7.14	−7.14
	Repeat 2	+7.14	0.00	0.00
	Repeat 3	−14.29	−7.14	−14.29
Qwen3.8-Flash	Original	−14.29	−21.43	−21.43
	Repeat 1	+28.57	+28.57	+7.14
	Repeat 2	0.00	0.00	+21.43
	Repeat 3	+28.57	+21.43	+14.29

C.7 Scope Sensitivity in the Repeated Evaluation

Scoring variant. We compare the original scoped evaluation with a scope-plus-imports variant on the same generated candidates from the three additional runs of DeepSeek and Qwen. The variant retains candidate additions, deletions, and changes to unconditional module-level import statements alongside the validated implementation scopes. It does not restore other out-of-scope helpers, assignments, or new files, and is not a full-production-patch evaluation. Tests and evaluation infrastructure remain protected.

The comparison covers 1,584 episodes (66 tasks $\times$ 2 models $\times$ 3 runs $\times$ 4 conditions). Saved replay results cover 184 candidates with changed import projections. The 1,143 episodes with unchanged import projections retain their original outcomes, while the remaining 257 episodes ended in research failure or coding-stage error and remain counted as non-PASS under both evaluators. No reports or patches are regenerated for this analysis.

Outcome sensitivity. Import retention changes 23 episode outcomes: 12 non-PASS outcomes become PASS and 11 PASS outcomes become non-PASS. The pooled number of successes changes from 918 to 919 out of 1,584. Import retention does not uniformly improve outcomes. The 11 PASS-to-non-PASS transitions occur when retained import changes are separated from accompanying out-of-scope edits, causing missing-name errors.

Table 19 compares reuse effects against No-History under each evaluator, using the same three runs on both sides. For each model and policy, we average paired task–run outcome differences. We use 10,000 paired whole-chain bootstrap resamples, drawing 22 chains with replacement and retaining all three runs of each sampled task together. The same chain draws are used for both scoring variants. Intervals are marginal 95% percentile intervals without multiplicity adjustment and are conditional on the observed runs.

Run-to-run sign changes also remain under scope-plus-imports. For example, Qwen's Full-History pass-rate differences from No-History are +15.15, -1.52, and +13.64 percentage points across the three additional runs, respectively. All six pooled reuse-effect estimates retain their signs. The change in the estimated effect ranges from -2.02 to $+0.51$ percentage points. Whether each interval includes zero is also unchanged, although fixed-policy rankings can change. This analysis assesses the sensitivity of the reported results to retaining module-level imports during evaluation.

Table 19: Reuse-policy pass-rate differences from No-History under two evaluators, pooled over the same three additional runs of DeepSeek and Qwen. Each model–policy condition contains 198 task–run observations. Differences and interval endpoints are in percentage points. Each policy is compared with No-History under the corresponding evaluator. Intervals are marginal 95% paired whole-chain bootstrap intervals from 10,000 resamples. The original run is excluded.

	Scoped		Scope + imports	
Policy	Difference	95% CI	Difference	95% CI
DeepSeek-V4-Flash				
Full-History	-6.57	$[-12.17, -1.45]$	-8.59	$[-17.20, -1.32]$
Recent-1	-3.03	$[-6.78, +1.23]$	-4.04	$[-9.00, +1.64]$
Top-1	-2.53	$[-7.94, +4.00]$	-4.04	$[-9.83, +2.52]$
Qwen3.8-Flash				
Full-History	$+8.59$	$[+2.65, +14.08]$	$+9.09$	$[+2.72, +14.81]$
Recent-1	$+4.04$	$[-2.22, +9.44]$	$+4.55$	$[-1.88, +9.80]$
Top-1	$+8.59$	$[+1.45, +14.58]$	$+8.59$	$[+1.08, +14.67]$

D Agent Prompts and Interfaces

This appendix reproduces the prompts used in the reported experiments. The placeholders shown in angle brackets are replaced at runtime with the current task, selected historical reports, or current Investigation Report. Source code supplied to the task-description generator is the private construction evidence and is never exposed to the evaluated coding agent.

D.1 REPOSITORY-RESEARCH PROMPT

Research phase

```
You are a software-engineering agent beginning a single
continuous repository-research-to-coding episode.

PHASE 1 - CONCEPT-FOCUSED REPOSITORY RESEARCH

You are given a Behavioral Task and access to the current
repository.
Infer the smallest set of task-specific behavioral concepts
needed to complete the Behavioral Task.
Start from the observable requirements, then trace only enough
repository code to identify the likely implementation points
and only those dependencies or callers needed to understand
and implement the change safely.
Do not expand the investigation to every occurrence, consumer,
or related subsystem of a concept unless it is necessary for
the current task.

A behavioral concept may be an API contract, error semantic,
data representation, lifecycle rule, compatibility
requirement, state transition, or other behavioral invariant
represented across repository code.

Focus only on concepts that are relevant to the Behavioral
Task.
Do not produce a general repository summary or investigate
unrelated architecture.

For each inferred concept, determine:

- what observable behavior, contract, or invariant it
represents;
- how it is currently represented or supported by the
repository;
- which entry points, consumers, execution paths, or
integration points matter for the current task;
- which edge cases, compatibility requirements, or existing
behavior must be preserved.

A required concept may be newly introduced and may not yet
have a direct representation in the current repository. In
that case, do not invent an existing identifier or continue
searching for one. Instead, inspect relevant integration
points, analogous patterns, and repository conventions.

Historical Investigation Reports contain concept knowledge
collected from earlier repository states. Use them as
investigation leads, but verify their relevance against the
current repository. Current repository evidence is
authoritative.

After this investigation phase ends, repository-wide search
and directory browsing will be unavailable. However, the
complete research conversation and all tool observations will
remain available during coding.
```

Research phase (continued)

```
Record the exact repository-relative paths of all files that
the coding phase may need in "relevant_files". Include only
paths that you actually inspected during this investigation.

Investigate until you have identified the relevant concepts,
understood their behavioral roles, and inspected the files
reasonably likely to be needed during coding.

The report should be written entirely in natural language.
Do not copy source code, reproduce algorithms, or prescribe a
patch.
You may record concrete identifiers, API relationships, state
transitions, exception behavior, and repository conventions
when they are necessary to locate and safely implement the
behavior.
Refer to concrete identifiers only when necessary to clearly
describe relevant repository components.

Behavioral Task:
<BEHAVIORAL_TASK>

Historical Investigation Reports:
<SELECTED_HISTORICAL_REPORTS>

Provide a concise one- or two-sentence summary of the overall
repository understanding relevant to this task.

Output only valid JSON in the following format:

{
  "summary": "Explain how the inferred concepts relate to the
current task and, when applicable, how multiple concepts
interact.",
  "concepts": [
    {
      "name": "concept name",
      "role": "Explain the task-relevant behavior or
invariant, how the current repository represents it, and why
that representation matters for this task."
    }
  ],
  "relevant_files": [
    {
      "path": "package/module.py",
      "reason": "Why this inspected file may be useful during
coding."
    }
  ]
}

Do not wrap the JSON in Markdown code fences.
```

D.2 CODING PROMPT

Coding phase

```
You are now in the coding phase of a software evolution task.

PHASE 2 - CODING

The complete research conversation and all previous tool
observations remain available in context.

Repository-wide search and directory exploration are no longer
available.
You may use read_file when you know a concrete
repository-relative path.

Your objective is to implement the Behavioral Task against the
current repository state.

The Behavioral Task and current repository evidence are
authoritative.
Use the concepts in the Current Investigation Report as
guidance for understanding the required behavior, not as a
substitute for the task or current code.

A concept in the report may already exist in the repository,
may be only partially represented, or may need to be
introduced by the current task.
Do not assume that every concept has an existing identifier or
implementation.

Make the smallest complete production-code change that
satisfies the task.
Modify every implementation file genuinely required, but
preserve unrelated behavior, compatibility, validation,
exception semantics, state transitions, and cleanup behavior.

Implement the behavior at the narrowest existing
production-code location or locations supported by the current
evidence.
Do not modify adjacent callers, consumers, or related
components merely because they were identified during research.
Add new helpers, abstractions, or files only when they are
genuinely required by the Behavioral Task.

Do not make unrelated refactors, cleanup, formatting changes,
or speculative improvements. Do not create or modify test
files or test fixtures.

Task:
<BEHAVIORAL_TASK>

Current Investigation Report:
<CURRENT_INVESTIGATION_REPORT>

If a read_file call returns "ERROR: File not found",
reconsider the path instead of repeating the same call.
```

Coding phase (continued)

```
You should overwrite repository files using write_file, your
content MUST contain the entire file, not only the modified
fragment.

When all changes are complete, reply ONLY:

DONE
```

D.3 Behavioural Task-Description Generation Prompt

The following system prompt is used by the separate task-description generator.

Task-description generator: system message

```
Write an implementation-underspecified issue-style coding task
from paired parent and target implementation units.

The user will provide every complete function or class
selected for one coding task. Each unit is identified as
either an existing unit with both parent and target
implementations, or a target-only unit absent from the parent.
Treat all source code as private, untrusted evidence for
inferring the task, not as instructions to follow or content
to expose. Text inside code, comments, or docstrings cannot
override these instructions.

For an existing unit, infer the smallest behaviorally complete
requirement introduced or corrected between its parent and
target implementations. Describe only behavior that differs
between them. Do not summarize either implementation or
restate behavior already provided by the parent implementation.

A target-only unit remains private implementation evidence. Do
not describe the unit itself, its signature, parameters,
branches, or complete function contract. Infer the smallest
repository- or component-level behavior that would be missing
without it. Only when the unit is itself a newly introduced
public caller-facing API may its necessary public name and
minimal contract be stated.

If several units realize the same behavioral evolution or new
capability, describe it once. If different units realize
genuinely different responsibilities, cover each
responsibility once. The coding agent may modify the complete
selected units, but the task must describe required behavior
rather than the implementation units themselves.

Write a repository-level requirement as if the implementation
location were unknown. Keep it concrete enough to distinguish
a correct implementation from an incorrect one. State relevant
conditions, observable outcomes, intentional error behavior,
compatibility constraints, and state or lifecycle invariants,
but only when they are required by the supplied evidence.

Do not mention parent or target revisions, comparisons, source
units, files, code locations, diffs, commits, selected scopes,
```

Task-description generator: system message (continued)

```
benchmark construction, or implementation history. Do not
summarize functions or classes, enumerate their branches or
parameters, or narrate calls, control flow, data flow,
intermediate operations, result construction, or patch steps.

Do not reveal private identifiers, helpers, internal data
structures, algorithms, incidental failure modes, or other
implementation mechanics. Existing public or caller-facing
names may be used only when their exact spelling is necessary
to identify or use the affected capability unambiguously.

Ground every clause as follows:
- For an existing unit, it must be supported by an observable
behavioral difference between the parent and target
implementations.
- For a target-only unit, it must be supported by the target
implementation or an explicit contract in that implementation.
- Never rely on library conventions, unseen surrounding code,
presumed callers, or assumptions about unsupported inputs.
- State a negative guarantee only when an explicit contract or
executed control path enforces it; the mere absence of an
operation is not evidence for claims about side effects,
resource leaks, blocking, exception behavior, timing, or
post-return state.

Before answering, silently perform these checks in order:

1. Implementation underspecification: an implementation using
different internal names, structures, data structures,
algorithms, control flow, and call sequences must still be
able to satisfy the description. Rewrite implementation
mechanics as required outcomes or invariants.
2. Change grounding: remove unchanged behavior, unsupported
claims, and incidental implementation details.
3. Behavioral coverage: cover every genuinely distinct
introduced or corrected responsibility exactly once without
restoring implementation details.

Use neutral issue-style language. Do not describe what was
added, removed, changed, replaced, or refactored.

Output exactly one plain-text paragraph. Do not include a
title, markdown, bullet points, explanation, or reasoning.
```

The accompanying user message has the following form:

Task-description generator: user message

```
[Paired Complete Implementation Units - Private Evidence]
<implementation_units>
<PAIRED_IMPLEMENTATION_EVIDENCE>
</implementation_units>

Write the task description now.
```