ANALYZING AND MITIGATING COST-INEFFICIENT BEHAVIORS IN CODING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Although effective, coding agents often incur substantial monetary costs. Their recurring *cost-inefficient behaviors* remain underexplored. We conduct a systematic study of behavioral cost inefficiencies in coding agents, analyzing 1,200 trajectories from Claude Code and Mini-SWE-Agent across four configurations on SWE-bench Verified. We identify three cost-inefficient behaviors: *subsumed retrieval*, *similar script generation*, and *test re-execution*. We then evaluate three mitigation strategies: structure-aware retrieval, agent-synthesized skills, and developer-designed skills, over 10k trajectories on held-out SWE-bench Verified and Pro tasks. Our main findings are: (1) The three behaviors affect 79.00–98.00% of coding tasks and account for up to 22.75% of task cost. (2) Structure-aware retrieval can introduce retrieval overhead and alter agent delegation, causing inconsistent improvements in retrieval efficiency and cost increases of up to 28.14%. (3) Agent-synthesized skills tend to produce low-level, trace-specific guidance, limiting their effectiveness and generality. (4) In contrast, developer-designed skills provide high-level, trace-agnostic guidance, reducing cost by up to 41.73%, roughly twice the maximum gain from agent-synthesized skills.

1 INTRODUCTION

Coding agents have become increasingly capable, with systems such as Claude Code, Codex, and Cursor combining LLMs with tools and reusable skills (Anthropic, 2025b; OpenAI, 2025; Anysphere, 2026). Yet, this capability comes at a steep cost. Token spending can reach millions of dollars annually (Bai et al., 2026), and Uber reportedly exhausted its annual AI coding budget within four months (Angelo, 2026). The pressure also affects model providers: subscribers can consume tens of thousands of dollars of model usage on a monthly plan, leaving OpenAI losing money on subscriptions and Anthropic rate-limiting Claude Code (TechCrunch, 2025; Zeff, 2025). In addition, higher spending does not reliably improve performance (Kapoor et al., 2024). These trends make the cost efficiency of coding agents a growing concern for both individual developers and enterprises.

Despite the effectiveness of existing cost-reduction approaches (Wang et al., 2026; Su et al., 2025; Liu et al., 2026a), recurring cost-inefficient behaviors during agent execution remain largely undetected, accumulating substantial end-to-end cost. Figure 1 illustrates this problem using Claude Code on SWE-bench Verified task `Django-13158` (Jimenez et al., 2024; Chowdhury et al., 2024), which requires fixing `QuerySet.none()` to return an empty result on combined querysets. Claude Code delegates code exploration to a subagent, retrieves additional context, applies a patch, and validates it with generated and repository tests. Although the task passes, three cost-inefficient behaviors account for 17.25% of its cost[1]. First, the main agent retrieves 20 lines from `query.py` and 85 lines from `compiler.py` already covered by subagent retrievals. This ***subsumed retrieval*** accounts for 6.96% of task cost. Second, it generates four similar inline test scripts sharing 21 lines of logic with minor variations. This ***similar script generation*** consumes another 7.10%. Third, it incorrectly executes `test_qs_combinators` eight times in the same way without updating the patch, repeatedly hitting the same module error. This ***test re-execution*** costs 3.19%. These observations motivate our first research question (**RQ1**): *What cost-inefficient behavior patterns occur in coding agents?*

Among the detected behaviors, subsumed retrieval is the most prevalent. A promising remedy is structure-aware retrieval (Ouyang et al., 2025b; Liu et al., 2025b; Chen et al., 2025), which enables

[1]Behavior cost is computed from the input, output, and cache tokens consumed by each flagged action using the corresponding model token prices (Appendix A.1.2).

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

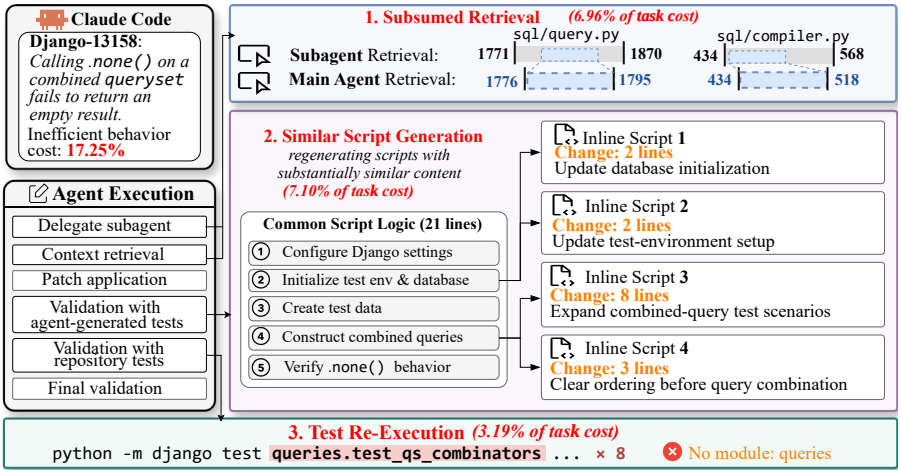


Figure 1: Cost-inefficient behaviors of Claude Code on SWE-bench Verified task `Django-13158`.

direct access to complete code entities and explicit dependencies, reducing broad-to-narrow re-reads that cause subsumed retrieval. Although prior work reports improved localization accuracy and efficiency, whether these benefits generalize across agent architectures and translate into end-to-end cost savings remains unclear. We therefore ask **RQ2**: *How does structure-aware retrieval affect cost-inefficient behaviors and the cost efficiency of coding agents?*

While structure-aware retrieval primarily targets retrieval-related inefficiencies, agent skills (Anthropic, 2025a) offer a lightweight intervention for addressing broader cost-inefficient behaviors. Such skills provide guidance that agents can apply adaptively throughout execution. We therefore study two complementary approaches: agent-synthesized skills and developer-designed skills. We ask **RQ3**: *How do **agent-synthesized skills** (SynSkills) and **developer-designed skills** (DevSkills) affect cost-inefficient behaviors and the cost efficiency of coding agents?*

To answer these RQs, we study cost-inefficient behaviors and their mitigation across four configurations: Claude Code (*CC*) with Sonnet 4.6 (*S46*) (Anthropic, 2026b), and Mini-SWE-Agent (*MSA*) (Yang et al., 2024) with *S46*, MiniMax-M3 (*MM3*) (MiniMax, 2026), or Qwen-3.5 Plus (*Q35+*) (Qwen Team, 2026). For **RQ1**, we analyze 1,200 trajectories on 300 SWE-bench Verified tasks. For **RQ2–RQ3**, we evaluate CodeGraph (Mchenry, 2026) as a representative structure-aware retrieval tool, alongside SynSkills and DevSkills, on 200 held-out Verified and 100 SWE-bench Pro tasks (Deng et al., 2025), with repeated runs to account for execution stochasticity.

Our paper makes the following contributions:

- We conduct a systematic study of behavioral cost inefficiencies in coding agents, analyzing 1,200 trajectories from four agent configurations on SWE-bench Verified. Our findings include:
  - We identify three recurring cost-inefficient behaviors: ***subsumed retrieval (SubRetrv)***, retrieving code fully covered by prior retrievals; ***similar script generation (SimScrpt)***, regenerating highly similar scripts with only minor changes; and ***test re-execution (ReTest)***, rerunning identical tests without an updated patch. Together, they cover 79.00–98.00% of tasks and account for 6.86–22.75% of task monetary cost.
  - ***SubRetrv*** is the most prevalent behavior, affecting up to 92.33% of tasks and 11.41% of cost. SubRetrv in Claude Code mainly stems from re-reading subagent-retrieved code; in Mini-SWE-Agent, it arises from low-feedback editing, zoom-in localization, and long debugging loops.
  - ***SimScrpt*** affects up to 68.00% of tasks and occurs 5.91–9.98× more often in Mini-SWE-Agent than in Claude Code. Built-in guidance in Claude Code largely confines SimScrpt to ephemeral scripts, while Mini-SWE-Agent also regenerates similar files for testing and editing.
  - ***ReTest*** affects 49.67–83.00% of tasks and contributes up to 5.39% of task cost. It primarily arises from repository-specific test knowledge gaps, test-signal recovery, and progress stalls.
- We address a key gap in the evaluation of structure-aware retrieval by examining whether it consistently improves retrieval efficiency and whether such gains translate into end-to-end cost savings.
  - **Structure-aware retrieval is not inherently cost-efficient.** Depending on agent architecture and model, its verbose feedback and altered agent delegation can introduce retrieval overhead, yielding inconsistent SubRetrv reductions and end-to-end cost increases of up to 28.14%.

- We design and evaluate a lightweight agent-driven framework that synthesizes behavioral optimization skills from diagnosed cost-inefficient behaviors.
  - **Agent-synthesized skills tend to produce low-level, trace-specific guidance, limiting their effectiveness and generality.** They yield no robust cost increase and reduce cost by at most 22.32%, roughly half the maximum achieved by developer-designed skills.
- We develop a compact set of behavior-targeted optimization skills. **Our high-level, trace-agnostic developer-designed skills achieve the largest cost savings**, reducing task cost by 7.88–41.73%, with maximum reductions of 38.57–88.91% across the three behaviors. These results highlight the value of human insights in agent behavioral optimization.

Our implementation and analysis are available in this anonymous repository.

2 Study Setup

Research Questions. For **RQ1**, we analyze 1,200 trajectories across four agent configurations and 300 tasks. We normalize trajectories and label actions using an author-defined taxonomy with rule-based and LLM-assisted labeling (Appendix A.1.1). We then define and manually calibrate detectors to identify three cost-inefficient behaviors and examine their underlying mechanisms. For **RQ2–RQ3**, we evaluate three approaches for mitigating the cost-inefficient behaviors, comparing each with its baseline on Pass@1, task monetary cost (cost), cost-of-pass (CoP) (Erol et al., 2026), and behavior changes. In **RQ2**, we equip agents with CodeGraph for structure-aware retrieval. *CC* accesses this through an MCP tool, while *MSA* uses command-line wrappers. For SynSkills in **RQ3**, we build a lightweight analysis agent using each configuration's backbone model to distill corrective rules from its RQ1 trajectories and apply Trace2Skill (Ni et al., 2026) to consolidate them into a configuration-specific skill set. For DevSkills, we manually derive a shared skill set from the RQ1 findings under two principles: (1) generalize across configurations and (2) mitigate cost-inefficient behaviors without sacrificing task performance. Following Trace2Skill, all skill sets are preloaded into the agent system prompt for evaluation (Appendix A.4).

Agent Configuration. To disentangle the effects of agent framework and backbone model on cost-inefficient behaviors, we study two agent frameworks and vary the model within one of them. *CC* is a proprietary, production-grade framework with structured `Read`/`Edit` tools and subagent delegation, whereas *MSA* is an open-source, single-agent framework that interacts with repositories through shell commands. We evaluate four agent configurations: *CC* with *S46* as the main model and Haiku 4.5 (*H45*) for subagents, and *MSA* with *S46*, *MM3*, or *Q35+*. Comparing *CC* with MSA_{S46} captures framework-associated differences under the same main model; comparing the three *MSA* configurations reveals model-associated differences under a fixed framework.

Benchmarks & Task Split. To separate behavior discovery from mitigation evaluation, we split the 500 *SWE-bench Verified* tasks within each repository by creation date: the earliest 300 tasks form the **RQ1** analysis set and the remaining 200 form the held-out set *Verified-200* for **RQ2–RQ3**, while preserving repository distribution. For cross-benchmark evaluation, we sample 100 tasks from *SWE-bench Pro* (*Pro-100*), covering all 11 repositories and favoring costlier tasks (Appendix A.1.3). Together, *Verified-200* and *Pro-100* form the 300-task mitigation evaluation set for **RQ2–RQ3**.

Controlling Execution Randomness. To account for run-to-run noise in **RQ2–RQ3**, we run eight baselines (four configurations $\times$ two benchmarks) three times and estimate noise floors using the standard deviation of Pass@1 and coefficients of variation of cost and CoP. For unreplicated approaches, we transfer the matching baseline's noise floor, following the standard equal-variance assumption that our mitigations do not alter the sampling and infrastructure nondeterminism driving run-to-run noise (Ouyang et al., 2025a). We validate this assumption on nine replicated approach cells spanning approaches, agent frameworks, and benchmarks; eight show no distinguishable noise amplification, while we use the larger observed noise for the exception (Appendix A.1.4). We classify an effect as *robust* at one-sided $p < 0.05$, imposing a stricter threshold on single-run effects. Because behavior-level metrics are noisier, we report their changes only as mechanistic evidence.

3 RQ1: Analyzing Cost-Inefficient Behaviors in Coding Agents

We identify **three** cost-inefficient behaviors across *information retrieval, script management, and solution validation* in coding agent execution. (1) **Subsumed Retrieval:** retrieving code fully covered by previous retrievals. (2) **Similar Script Generation:** regenerating highly similar scripts with minor changes. (3) **Test Re-Execution:** re-executing identical tests without updated patches.

Table 1: Prevalence, frequency, and cost contribution of cost-inefficient behaviors across agent configurations. **Tasks**: percentage of tasks exhibiting the behavior; **Freq.**: average number of behaviors per task; **Cost**: average share of per-task monetary cost attributable to the behavior.

	CC			MSA_{S46}			MSA_{MM3}			$\text{MSA}_{\text{Q35+}}$		
Behaviors	**Tasks**	**Freq.**	**Cost**	**Tasks**	**Freq.**	**Cost**	**Tasks**	**Freq.**	**Cost**	**Tasks**	**Freq.**	**Cost**
SubRetrv	64.33%	2.15	5.01%	87.33%	3.45	8.42%	92.33%	5.14	7.88%	89.33%	6.37	11.41%
SimScrpt	20.67%	0.43	1.02%	51.33%	2.54	9.57%	68.00%	4.29	8.85%	57.67%	3.20	7.91%
ReTest	49.67%	2.09	0.83%	69.00%	2.51	3.17%	83.00%	5.29	5.39%	66.00%	2.10	3.43%
Total	**79.00%**	**4.67**	**6.86%**	**97.33%**	**8.50**	**21.16%**	**98.00%**	**14.72**	**22.12%**	**96.67%**	**11.68**	**22.75%**

Table 1 reveals that the three behaviors are both prevalent and costly, collectively affecting at least 79.00% of tasks and accounting for up to 22.75% of task cost, and *CC* consistently exhibits the lowest overall level of inefficiency (Appendix A.2.1). Figure 2 further shows their temporal patterns: SubRetrv concentrates in the first half of the trajectory, coinciding with the retrieval-heavy stage; ReTest shifts toward the middle-to-late stages, where the trajectory becomes more test-heavy for patch validation; and SimScrpt is distributed more broadly rather than confined to a specific phase. We next formalize each behavior and analyze its detection, severity, and underlying mechanisms.

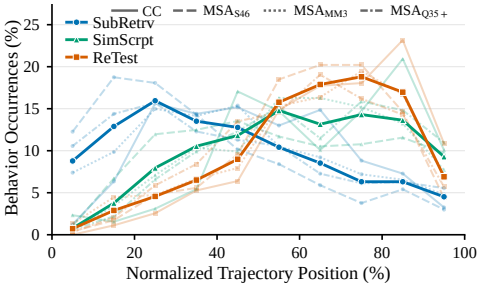


Figure 2: Temporal distribution of the cost-inefficient behaviors. Bold curves aggregate configurations; thin curves show individual ones.

3.1 Subsumed Retrieval

Detection and Severity. For each retrieval b returning at least five non-empty lines of content, we scan preceding retrieval actions backward and identify the first action a whose returned context fully covers that of b. We treat (a, b) as a SubRetrv pair and flag b as one SubRetrv action. Table 1 shows that SubRetrv is the most prevalent cost-inefficient behavior, affecting 64.33–92.33% of tasks, occurring 2.15–6.37 times per task, and accounting for 5.01–11.41% of task cost.

Underlying Mechanisms. To study when and why SubRetrv occurs and how its mechanisms differ between *CC* and *MSA*, we classify each SubRetrv action into different scenarios in Table 2. The results reveal substantial differences across agent architectures. *Cross-Agent SubRetrv*, where a main-agent retrieval is subsumed by a previous subagent retrieval, is unique to *CC* and accounts for 50.15% of its SubRetrv. *CC* offloads exploration to subagents, which return summaries rather than the retrieved code. So the main agent may re-retrieve the same region when needed later, as illustrated in Figure 1.

Table 2: SubRetrv actions by scenario, assigned in top-to-bottom priority order. Subscripts indicate the percentage of SubRetrv actions.

	SubRetrv Actions			
Occurrence Scenario	**CC**	MSA_{S46}	MSA_{MM3}	$\text{MSA}_{\text{Q35+}}$
Cross-Agent SubRetrv	$\mathbf{324}_{\mathbf{50.15\%}}$	—	—	—
Patch-Adjacent SubRetrv	$56_{8.67\%}$	$274_{26.47\%}$	$277_{17.96\%}$	$374_{19.56\%}$
Within-Sequence SubRetrv	$171_{26.47\%}$	$\mathbf{311}_{\mathbf{30.05\%}}$	$255_{16.54\%}$	$\mathbf{897}_{\mathbf{46.91\%}}$
Long-Distance SubRetrv	$73_{11.30\%}$	$275_{26.57\%}$	$\mathbf{808}_{\mathbf{52.40\%}}$	$472_{24.69\%}$
Other	$22_{3.41\%}$	$175_{16.91\%}$	$202_{13.10\%}$	$169_{8.84\%}$

Patch-Adjacent SubRetrv occurs when the SubRetrv immediately precedes or follows a patch. It is more common in *MSA* (274–374 actions) than in *CC* (56) due to different editing interfaces. *MSA*'s shell-based editing, such as `sed -i` or generated scripts, may provide little feedback or fail silently, prompting agents to re-read code for localization or post-edit inspection. In contrast, *CC*'s retrieval tools provide line numbers, while its editing tools report detailed edit outcomes, including the updated code, reducing such scenarios.

Within-Sequence SubRetrv occurs when both retrievals in a SubRetrv pair fall within an uninterrupted retrieval sequence. It typically follows a locate-then-zoom pattern, where the agent first reads a broad region and then retrieves a subregion for precise localization. This is common in *MSA*, where commands such as `cat` omit line numbers and often trigger follow-up `grep -n` or `sed -n` calls; *CC*'s `Read` already returns line-numbered code, reducing this need (171 vs. 255–897 actions).

Long-Distance SubRetrv occurs when both retrievals in a SubRetrv pair are separated by at least 10 steps. Manual inspection shows that it typically reflects longer debugging loops, where earlier

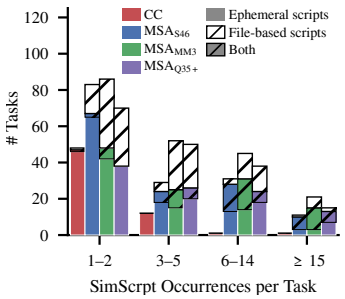


Figure 3: Distribution of SimScrpt occurrences per task.

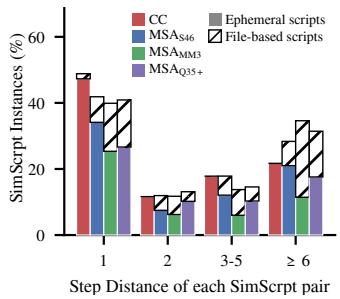


Figure 4: Distribution of SimScrpt pair step distances.

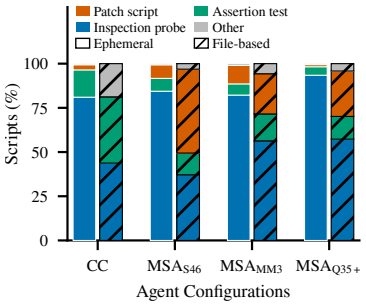


Figure 5: Functions of ephemeral and file-based SimScrpt scripts.

context becomes less salient and is re-retrieved. It is most prominent in *MSA$_{MM3}$* (52.40%), which is consistent with its longest average trajectory length (77.99 steps vs. 29.99–58.06 for others).

> **Finding 1.** *Subsumed Retrieval* is the most prevalent behavior, affecting 64.33–92.33% of tasks and contributing 5.01–11.41% of task cost. Its causes are architecture-dependent: Cross-Agent SubRetrv dominates in *CC*, while *MSA* mainly suffers from low-feedback editing, zoom-in localization, and long debugging loops.

3.2 Similar Script Generation

Behavior Detection. We define SimScrpt as regenerating near-duplicate scripts with minor differences rather than editing existing scripts, wasting output tokens and forgoing artifact reuse. We extract *ephemeral scripts* (e.g., `python -c` and heredocs) and *file-based scripts* (`.py` files) from agent actions, remove blank and comment-only lines, and retain scripts with at least five lines. For each generation action b, we identify the nearest preceding action a whose script has line-level Jaccard similarity[2] of at least 0.60. We treat (a, b) as a SimScrpt pair and count b as one occurrence.

Prevalence and Severity. Table 1 shows that SimScrpt affects up to 68.00% of tasks and contributes up to 9.57% of task cost. Interestingly, SimScrpt is substantially more severe in *MSA*, affecting 51.33–68.00% of tasks and contributing 7.91–9.57% of task cost, versus 20.67% and 1.02% in *CC*. It also occurs 5.91–9.98× more frequently in *MSA* (2.54–4.29 vs. 0.43). Figure 3 further reveals a pronounced high-frequency tail in *MSA*: 42–66 tasks per configuration exhibit at least six SimScrpt occurrences, compared with only two in *CC*.

Underlying Mechanisms. *When and why SimScrpt occurs.* SimScrpt arises during issue reproduction, code editing, and patch validation, explaining its broad distribution in Figure 2. Figure 4 further reveals two patterns: 39.89–48.84% of occurrences immediately follow a similar script, typically reflecting minor revisions with full-script regeneration, while 21.71–34.60% occur at a step distance of at least six. The underlying causes differ by script form. Ephemeral scripts leave no reusable artifact, requiring agents to regenerate similar code when needed again. File-based scripts persist on disk, but become less salient to the agent as its trajectory grows. Reusing them requires re-reading files for precise editing, while shell-based edits such as `sed -i` are error-prone, as discussed in *Patch-Adjacent SubRetrv*. Regenerating a near-duplicate script can therefore become the easier immediate choice.

Ephemeral vs. file-based SimScrpt scripts. To understand their differences, we classify SimScrpt scripts by their functionalities using abstract syntax tree (AST) analysis (Figure 5). A *patch script* modifies source files; an *assertion test* performs explicit pass/fail validation; and an *inspection probe* queries or prints runtime information, such as object values, attributes, or program state. Scripts are assigned by priority: patch, test, then probe. Ephemeral SimScrpt scripts overwhelmingly consist of inspection probes (81.03–93.45%). Agents commonly use `python -c` and heredocs as disposable probes: they generate a script, inspect certain output, and discard it, then regenerate a similar probe when the information is needed again. File-based SimScrpt scripts are more functionally diverse: inspection probes decrease, while assertion tests and, in *MSA*, patch scripts account for larger

[2]We select 0.60 by manually inspecting 20 SimScrpt pairs per configuration. Lower thresholds often match generic scaffolding, while 0.60 retains highly similar task-specific logic.

shares. Because these files persist and encode reusable validation or editing logic, regenerating them represents a clearer missed opportunity for artifact reuse.

Why SimScrpt script forms differ across architectures. Figures 3 and 4 show that *CC* primarily exhibits ephemeral SimScrpt, whereas *MSA* exhibits both script forms. This difference is consistent with *CC*'s built-in instruction: "NEVER create files unless they're absolutely necessary for achieving your goal." This discourages persistent script creation and shifts toward ephemeral generation, whereas *MSA* has no such constraint. Patch scripts are rare in *CC* because its built-in `Edit` and `Write` tools directly support modification action, whereas *MSA*'s poor feedback of shell-based editing more often triggers agents to generate their own editing scripts.

> **Finding 2.** *Similar Script Generation* affects up to 68.00% of tasks and contributes up to 9.57% of task cost, occurring 5.91–9.98× more frequently in *MSA* than in *CC*. *CC*'s built-in guidance largely confines SimScrpt to ephemeral scripts, whereas *MSA* also repeatedly regenerates file-based testing and editing scripts. These patterns suggest that an ideal agent should persist reusable logic and use inline scripts for simple one-off probes.

3.3 Test Re-Execution

Detection and Severity. We partition trajectories into inter-patch windows because a new patch may justify rerunning tests, whereas repeated executions without a patch update validate the same code state. Within each window, we group executions by the executed tests, retain the final execution as potentially decision-relevant, and flag preceding ones as ReTest. *Table 1 shows that ReTest affects 49.67–83.00% of tasks and contributes up to 5.39% of task cost.* MSA_{MM3} has the highest prevalence and frequency, averaging 5.29 occurrences per task, over twice that of any other configuration.

Underlying Mechanisms. We identify three common causes of ReTest. (1) *Repository-specific test knowledge gaps* arise when agents are unfamiliar with the proper test runner or configuration, causing repeated failures. (2) *Test-signal recovery* occurs when test output is truncated, ambiguous, or poorly captured by the agent, prompting reruns to recover debugging information. (3) *Progress stalls* arise when agents enter long reasoning loops without updating the patch, repeatedly reconsidering the same diagnosis and rerunning tests despite sufficient prior signals. Such behavior reflects stalled decision-making without advancing the solution.

> **Finding 3.** *Test Re-Execution* affects 49.67–83.00% of tasks and accounts for up to 5.39% of task cost, with MSA_{MM3} exhibiting the highest prevalence and frequency. ReTest commonly arises from repository-specific test knowledge gaps, test-signal recovery, and progress stalls.

More detection details, analysis, and examples for RQ1 are provided in Appendix A.2.

4 Mitigating Cost-Inefficient Behaviors in Coding Agents

This section evaluates the effectiveness of the three mitigation approaches in optimizing cost efficiency and the identified cost-inefficient behaviors (Table 3). Figure 6 further relates changes in behavior-attributed cost to task cost. We first compare the three approaches across all settings, then examine each in depth.

4.1 Cross-Approach Comparison

We compare all three approaches across agent configurations and benchmarks. Three patterns emerge. **First, mitigation effectiveness generally increases from CodeGraph to agent-synthesized skills to developer-designed skills.** CodeGraph reduces SubRetrv inconsistently and even increases cost robustly in four of eight settings. SynSkills achieve moderate savings, whereas DevSkills produce the broadest reductions in both task cost and diagnosed behaviors. **Second, mitigation benefits depend on agent architecture and benchmark.** Improvements are generally larger under *MSA*, while *CC*'s system prompt and tool abstractions already suppress several inefficient behaviors, leaving less room for improvement. All approaches also perform better overall on Verified-200 than Pro-100 in identified behaviors. For skill-based approaches, this partly reflects a generalization challenge, as the skills are derived from Verified trajectories and applied to Pro. **Third, task cost changes are amplified relative to the changes in behavior-attributed cost.** In Figure 6, the fitted slopes are 2.83 on Verified-200 and 1.33 on Pro-100. Two effects can amplify the savings. (1) Preventing an inefficient action shortens the trajectory, reducing repeated cache-reading

Table 3: Effects of the three approaches on Pass@1, average cost, CoP, and the occurrence of identified behaviors. Subscripts show changes from baseline. For Pass@1, cost, and CoP, blue/red indicate robust improvements/regressions, while uncolored cells indicate changes within noise; for behaviors, blue/red indicate fewer/more instances.

		Verified-200				Pro-100			
Approach	Metric	CC	MSA_{S46}	MSA_{MM3}	MSA_{Q35+}	CC	MSA_{S46}	MSA_{MM3}	MSA_{Q35+}
CodeGraph	Pass@1 (pp)	$74.17_{-0.83}$	$72.50_{+0.00}$	$75.50_{+0.33}$	$72.50_{+5.33}$	$55.17_{+1.92}$	$51.72_{+1.92}$	$57.47_{-1.15}$	$39.08_{-3.45}$
	Cost ($)	$0.596_{+8.30\%}$	$0.619_{-3.03\%}$	$0.591_{+28.14\%}$	$0.112_{-3.29\%}$	$1.061_{+12.19\%}$	$1.132_{-0.25\%}$	$0.985_{+8.39\%}$	$0.150_{+12.99\%}$
	CoP	$0.804_{+10.22\%}$	$0.854_{-3.04\%}$	$0.782_{+27.52\%}$	$0.155_{-10.63\%}$	$1.923_{+8.69\%}$	$2.188_{-4.32\%}$	$1.715_{+10.20\%}$	$0.388_{+23.20\%}$
	SubRetrv	$0.16_{-84.18\%}$	$2.87_{-25.33\%}$	$6.36_{+18.01\%}$	$4.65_{-24.25\%}$	$0.38_{-75.72\%}$	$2.90_{+3.33\%}$	$6.34_{+34.74\%}$	$4.51_{+13.01\%}$
	SimScrpt	$0.26_{-25.85\%}$	$2.90_{-17.01\%}$	$5.90_{+28.49\%}$	$2.68_{-39.61\%}$	$0.10_{-0.21\%}$	$1.64_{+63.26\%}$	$1.99_{-1.33\%}$	$0.79_{-53.46\%}$
	ReTest	$1.75_{+13.85\%}$	$2.05_{-8.07\%}$	$5.55_{+7.34\%}$	$2.21_{+0.30\%}$	$0.56_{-0.89\%}$	$2.13_{+11.27\%}$	$4.49_{+25.45\%}$	$0.78_{+6.90\%}$
SynSkills	Pass@1 (pp)	$74.50_{-0.50}$	$72.00_{-0.50}$	$75.50_{+0.33}$	$69.67_{+2.50}$	$55.00_{+0.67}$	$52.00_{+0.00}$	$62.00_{+2.00}$	$46.00_{+1.00}$
	Cost ($)	$0.541_{-1.38\%}$	$0.496_{-22.32\%}$	$0.462_{+0.15\%}$	$0.115_{-0.70\%}$	$0.826_{-8.86\%}$	$1.006_{-6.13\%}$	$0.766_{-11.90\%}$	$0.132_{-2.22\%}$
	CoP	$0.726_{-0.12\%}$	$0.689_{-21.79\%}$	$0.611_{-0.33\%}$	$0.166_{-4.40\%}$	$1.503_{-9.48\%}$	$1.936_{-6.40\%}$	$1.235_{-15.09\%}$	$0.286_{-4.51\%}$
	SubRetrv	$0.56_{-44.62\%}$	$3.40_{-11.38\%}$	$4.17_{-22.56\%}$	$5.83_{-5.05\%}$	$0.98_{-34.31\%}$	$2.67_{+2.50\%}$	$3.76_{-14.29\%}$	$3.86_{+3.21\%}$
	SimScrpt	$0.20_{-41.85\%}$	$1.92_{-45.10\%}$	$3.98_{-13.43\%}$	$3.66_{-17.63\%}$	$0.08_{-11.28\%}$	$1.60_{+79.32\%}$	$1.19_{-35.44\%}$	$1.41_{-5.87\%}$
	ReTest	$1.75_{+13.20\%}$	$1.46_{-34.53\%}$	$4.67_{-9.86\%}$	$2.03_{-7.58\%}$	$0.62_{+25.44\%}$	$1.22_{-26.88\%}$	$3.19_{+0.84\%}$	$0.78_{+22.80\%}$
DevSkills	Pass@1 (pp)	$73.50_{-1.50}$	$73.00_{+0.50}$	$73.00_{-2.17}$	$69.83_{+2.67}$	$52.00_{-2.33}$	$52.00_{+0.00}$	$62.00_{+2.00}$	$46.00_{+1.00}$
	Cost ($)	$0.473_{-13.94\%}$	$0.372_{-41.73\%}$	$0.387_{-16.03\%}$	$0.103_{-11.34\%}$	$0.894_{-1.51\%}$	$0.733_{-33.00\%}$	$0.801_{-7.88\%}$	$0.140_{+3.57\%}$
	CoP	$0.643_{-11.83\%}$	$0.510_{-42.13\%}$	$0.530_{-13.61\%}$	$0.148_{-14.83\%}$	$1.718_{+3.58\%}$	$1.409_{-33.19\%}$	$1.291_{-11.21\%}$	$0.305_{+1.28\%}$
	SubRetrv	$0.35_{-64.89\%}$	$1.86_{-51.52\%}$	$4.05_{-24.82\%}$	$5.08_{-17.25\%}$	$0.60_{-59.78\%}$	$1.47_{-43.57\%}$	$4.02_{-8.36\%}$	$4.23_{+12.92\%}$
	SimScrpt	$0.15_{-54.93\%}$	$1.18_{-66.17\%}$	$4.02_{-12.45\%}$	$4.42_{-0.49\%}$	$0.01_{-88.91\%}$	$0.55_{-38.36\%}$	$1.85_{+0.36\%}$	$1.43_{-4.02\%}$
	ReTest	$1.47_{-4.75\%}$	$1.37_{-38.57\%}$	$3.62_{-30.05\%}$	$2.04_{-7.13\%}$	$0.72_{+45.68\%}$	$1.28_{-23.29\%}$	$3.00_{-5.16\%}$	$0.66_{+3.71\%}$

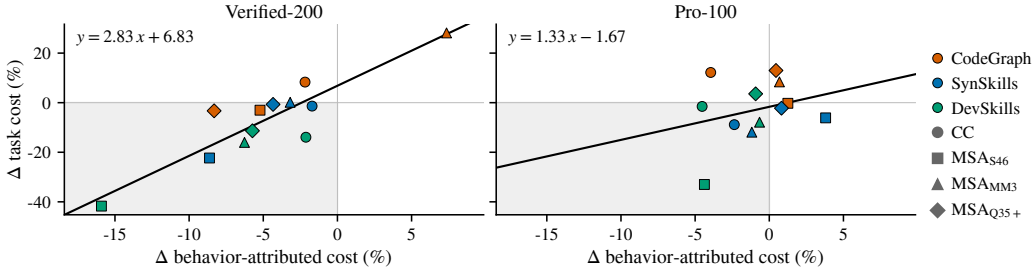


Figure 6: Changes in average task cost versus behavior-attributed cost, both relative to baseline task cost. The shaded quadrant indicates joint improvement, and the lines show the fitted linear trends.

of earlier context in later LLM calls. (2) It can also prevent additional unflagged actions triggered by certain cost-inefficient behaviors. For example, eliminating a ReTest can also avoid subsequent reasoning that interprets its output or diagnoses the same failure.

4.2 RQ2: Effectiveness of Structure-aware Retrieval

Table 3 shows that **CodeGraph neither consistently reduces SubRetrv nor improves task cost efficiency**.[3] Under *CC*, SubRetrv drops by over 75% on both benchmarks, yet cost rises by 8.30% on Verified-200 and robustly by 12.19% on Pro-100. Under *MSA*, SubRetrv increases in four of six settings. Overall, CodeGraph yields no robust cost reduction but four robust increases of 8.39–28.14%, and Pass@1 remains largely unchanged except for MSA_{Q35+} on Verified-200 (+5.33 pp). Thus, substantial SubRetrv reduction does not guarantee lower end-to-end cost.

Table 4 explains why. **For *CC*, verbose CodeGraph feedback and reduced use of cheaper subagents offset the savings from fewer SubRetrv actions**. CodeGraph cuts LLM and tool calls by 19.72–23.21% and 26.61–31.47%, respectively, but each query returns 8.2–16.6× more tokens than other retrieval actions. Total token usage therefore remains nearly unchanged. Meanwhile, *H45* subagent calls drop from 4.81 and 11.03 per task to zero on both benchmarks, while *S46* main-agent calls change only slightly (−0.42% and +8.17%). Thus, roughly the same token volume shifts to *S46*, whose token price is 3× that of *H45*, increasing *CC*'s cost by 8.30% on Verified-200 and robustly by 12.19% on Pro-100.

For *MSA*, cost rises because agents use CodeGraph additively, stacking CodeGraph queries and their verbose feedback on top of ordinary retrieval rather than replacing it. For MSA_{MM3}

[3]CodeGraph cannot run on 13 Pro-100 tasks (Appendix A.3.1). Its Pro-100 results are therefore measured and compared on the remaining 87 tasks.

Table 4: Effects of CodeGraph on average task token usage, average output tokens per CodeGraph and other retrieval call, and LLM/tool calls. Subscripts show percentage changes from the baselines.

	Verified-200				Pro-100			
Metric	CC	MSA_{S46}	MSA_{MM3}	MSA_{Q35+}	CC	MSA_{S46}	MSA_{MM3}	MSA_{Q35+}
Task tokens (M)	$0.91_{-1.70\%}$	$1.09_{-0.37\%}$	$4.13_{+29.29\%}$	$1.63_{-1.15\%}$	$1.70_{-0.36\%}$	$2.20_{-0.32\%}$	$7.28_{+9.11\%}$	$2.39_{+13.39\%}$
Output tokens / call	$777_{+140.96\%}$	$233_{+5.98\%}$	$251_{+5.94\%}$	$292_{-3.39\%}$	$955_{+55.74\%}$	$354_{+9.81\%}$	$302_{-5.64\%}$	$370_{-5.56\%}$
CodeGraph	5,199	876	1,205	1,122	5,273	1,111	1,007	1,111
Other retrieval	313	258	277	350	643	395	789	541
LLM calls	$19.91_{-19.72\%}$	$40.79_{-7.22\%}$	$95.39_{+13.60\%}$	$60.15_{-2.94\%}$	$29.18_{-23.21\%}$	$56.17_{-5.27\%}$	$143.91_{+8.20\%}$	$71.05_{+5.68\%}$
Main agent (S46)	$19.91_{-0.42\%}$	–	–	–	$29.18_{+8.17\%}$	–	–	–
Subagent (H45)	$0.00_{-100.00\%}$	–	–	–	$0.00_{-100.00\%}$	–	–	–
Tool calls	$19.43_{-26.61\%}$	$42.98_{-8.44\%}$	$94.66_{+12.97\%}$	$59.57_{-3.87\%}$	$32.74_{-31.47\%}$	$66.94_{-5.63\%}$	$143.37_{+8.07\%}$	$71.52_{+6.39\%}$
CodeGraph	2.23	5.07	6.75	6.31	3.60	8.13	6.33	5.65
Other retrieval	$7.16_{-53.66\%}$	$14.20_{-34.13\%}$	$35.38_{-2.92\%}$	$22.30_{-23.22\%}$	$15.68_{-53.68\%}$	$35.34_{-23.87\%}$	$75.01_{+3.16\%}$	$39.34_{-3.44\%}$

on both benchmarks and MSA_{Q35+} on Pro-100, ordinary retrieval remains nearly unchanged while agents add 5.65–6.75 CodeGraph calls per task; these extra calls and their verbose outputs raise total token usage by 9.11–29.29% and cost robustly by 8.39–28.14%. In the remaining settings, agents do replace ordinary retrieval with CodeGraph queries: ordinary retrieval drops by over 20%, but the verbose CodeGraph feedback keeps total token usage nearly unchanged, leaving cost savings indistinguishable from the baseline.

> **Finding 4.** Structure-aware retrieval, as exemplified by CodeGraph, is not inherently cost-efficient. Verbose feedback and tool integration can add retrieval overhead and alter agent delegation, leading to inconsistent SubRetrv reduction and higher cost. It should therefore be evaluated with explicit attention to feedback volume, tool integration, and model orchestration.

More details on CodeGraph integration and usage are provided in Appendix A.3.

4.3 RQ3: Agent-Synthesized vs. Developer-Designed Skills

Unlike CodeGraph, which changes how agents access code and targets only retrieval inefficiencies, skills can address broader cost-inefficient behaviors. SynSkills are configuration-specific sets of 23–41 operational rules. They target the three cost-inefficient behaviors through concrete, scenario-specific instructions, such as handling failed string substitutions, shell quoting issues, and permission errors. In contrast, DevSkills are seven general behavioral principles shared by all configurations. They require agents to state a concrete hypothesis before retrieval, reuse available context, and justify necessary re-reads (SubRetrv); to persist and revise scripts instead of creating similar variants (SimScrpt); to understand the test harness, capture test output, and rerun tests only when code changes or new evidence warrant it (ReTest); and to break unproductive loops (general).

DevSkills are substantially more effective and consistent than SynSkills. SynSkills robustly reduce average cost in three of eight settings by 8.86–22.32%, with no robust cost increase or Pass@1 degradation (Table 3). In comparison, DevSkills robustly reduce cost in six settings by 7.88–41.73%, nearly twice the maximum saving from SynSkills, with only one small Pass@1 drop (-1.50 pp for *CC* on Verified-200). The behavior-level results show the same pattern. For example, DevSkills reduce all three behaviors in MSA_{S46} on both benchmarks, including SubRetrv by 51.52% and 43.57%, compared with 11.38% and a 2.50% increase under SynSkills.

One possible mechanism is that DevSkills encode higher-level, trace-agnostic guidance, whereas SynSkills retain low-level, trace-specific instructions. For example, to mitigate SimScrpt, the MSA_{S46} SynSkills specify when to use ephemeral scripts or overwrite scripts, and how to handle non-ASCII shell content. Such rules depend on specific tasks and repositories, limiting generalization. In contrast, DevSkills cover the same cases with one principle: persist and reuse artifacts rather than regenerate similar variants. The principle applies to any task, which helps explain the stronger and more consistent effectiveness of DevSkills and the value of human abstraction in behavioral optimization.

> **Finding 5.** Agent-synthesized skills tend to produce low-level, trace-specific guidance, limiting their effectiveness and generality. In contrast, developer-designed skills provide high-level, trace-agnostic guidance and achieve broader cost savings, robustly reducing cost by 7.88–41.73% in six of eight settings.

More analysis and specific skill content for RQ3 are in Appendix A.4.

5 Related Work

Agent Trajectory Analysis. Observability platforms support execution tracing and cost monitoring (LangChain, 2026; Langfuse, 2026; AgentOps, 2026). Trajectory analyses further examine how agents approach tasks, where execution goes wrong, and how costs accumulate (Ceka et al., 2025; Bouzenia & Pradel, 2025; Majgaonkar et al., 2025; Zhang et al., 2025; Kim et al., 2025; Bai et al., 2026; Salim et al., 2026), identifying issues such as repeated actions and loss of coherence (Liu et al., 2026a; Kim et al., 2026). However, how recurring inefficient behaviors contribute to agent costs, why they occur, and whether they can be mitigated remain underexplored. To address this gap, we define and detect three cost-inefficient behaviors and analyze their prevalence, monetary impact, and underlying mechanisms across coding agents.

Agent Cost Reduction. Existing approaches improve agent cost efficiency through four strategies. Context compression and interaction reduction decrease the token usage of each step and the number of interactions (Gao & Peng, 2025; Pan et al., 2025; Nayab et al., 2026; Wang et al., 2026; Luk et al., 2026; Xiao et al., 2026; Li et al., 2026c; Jia et al., 2026). Agent orchestration allocates work across suitable models and tools (Shekhar et al., 2024; Wang et al., 2025; Su et al., 2025; Liu et al., 2025a; Ong et al., 2024). Runtime supervision intervenes when agents make errors, enter loops, or deviate from expected workflows (Lin et al., 2026; Liu et al., 2026a). However, their effects on specific cost-inefficient behaviors remain underexplored. We complement them by using diagnosed behaviors to guide optimizations and evaluating their effects on these behaviors, cost, and task performance. A fourth strategy improves agent toolsets. Structure-aware retrieval, for example, uses code graphs to guide repository navigation (Ouyang et al., 2025b; Liu et al., 2025b). Despite reported gains in localization accuracy and efficiency (Chen et al., 2025), whether these gains consistently translate into end-to-end cost savings remains unclear. We therefore examine its effects on retrieval efficiency and task cost across agent configurations.

Agent Skills. Agent skills (Anthropic, 2025a) provide reusable instructions and optional resources to guide task executions. Recent systems build skills from execution experience (Ni et al., 2026; Alzubi et al., 2026; Ma et al., 2026; Yang et al., 2026; Xia et al., 2026b; Liu et al., 2026b), incorporating reinforcement learning (Xia et al., 2026a; Li et al., 2026b), across web, multimodal, and generalist-agent settings (Zheng et al., 2025; Jiang et al., 2026; Zhou et al., 2026). Prior work evaluates skill effectiveness (Li et al., 2026a) and optimizes skills for both task success and inference cost (Gong et al., 2026). We derive skills from diagnosed cost-inefficient behaviors and compare our agent-synthesized and developer-designed skills in terms of agent cost efficiency.

6 Threats to Validity

Agent and Task Coverage. Budget limits our study to four agent configurations, SWE-bench Verified, and 100 SWE-bench Pro tasks. Behaviors and intervention effects may differ elsewhere. **Behavior Identification.** Parsing, labeling, and detection mechanisms may introduce detection errors despite explicit implementations and manual calibration. **Cost and Outcome Measurement.** We measure LLM inference monetary cost, excluding execution time and hardware costs. Pass@1 may miss regressions beyond test satisfaction. **Execution Randomness.** We triplicate baselines and nine intervention settings; budget limits others to single runs. Robustness estimates rely on baseline-derived variability, which may not fully establish statistical significance. **Structure-Aware Retrieval.** CodeGraph's popularity and robust structural retrieval implementations motivate its selection, but this may limit generalization to other tools. **Skill Design and Scope.** Findings depend on our agent-synthesis pipeline and developer-skill design. Script-based or other skill formats may differ from our prompt-based interventions.

7 Conclusions & Future Work

We analyze Claude Code and Mini-SWE-Agent trajectories on SWE-bench Verified, identifying three prevalent cost-inefficient behaviors: subsumed retrieval, similar script generation, and test re-execution. Evaluating three mitigation approaches on Verified and Pro shows that structure-aware retrieval can increase cost due to its verbose feedback and altered agent delegation. Agent-synthesized skills offer low-level, trace-specific guidance and moderate savings, while developer-designed skills provide high-level, trace-agnostic guidance and achieve the strongest and most consistent savings. Future work will extend this analysis to longer-horizon, interactive tasks and more diverse agents, examining inefficiencies beyond monetary cost. One could also use behavior diagnoses to guide optimizations and generate more efficient trajectories for agent training.

8 AI USE STATEMENT

AI within our experimental methods. Analyzing and mitigating the cost-inefficient behaviors of the coding agents is the object of our study, and their benchmark execution trajectories are used as our experimental data. We combine rule-based parsing with LLM-based action labeling under an author-defined taxonomy (Section 2). The authors design the labeling rules and thresholds, manually calibrate them, and examine their application to agent trajectories. In RQ3 (Section 4.3), an LLM-based pipeline generates agent-synthesized skills as a mitigating approach for evaluation.

AI assistance in research and writing. We use generative AI assistants to help debug and refactor author-written code, refine manuscript structure and wording, and improve table and figure layouts. The authors independently design the research questions, metrics, detection rules, analysis, findings, and conclusions. We review the AI-assisted work and take responsibility for the final manuscript, code, analysis, and results.

REFERENCES

AgentOps. Agentops documentation. https://docs.agentops.ai/v2/introduction, 2026. Accessed: 2026-08.

Salaheddin Alzubi, Noah Provenzano, Jaydon Bingham, Weiyuan Chen, and Tu Vu. EvoSkill: Automated skill discovery for multi-agent systems, 2026. URL https://arxiv.org/abs/2603.02766.

Jake Angelo. Uber burned through its entire 2026 AI budget in four months. now its COO is questioning whether it's worth it. https://fortune.com/2026/05/26/uber-coo-ai-spending-tokens-claude-code/, May 2026. Fortune. Accessed: 2026-08.

Anthropic. Agent skills. https://platform.claude.com/docs/en/agents-and-tools/agent-skills/overview, 2025a. Accessed: 2026-08.

Anthropic. Claude Code: Anthropic's agentic coding tool. https://claude.com/product/claude-code, 2025b. Accessed: 2026-08.

Anthropic. Claude Code v2.1.169. https://www.npmjs.com/package/@anthropic-ai/claude-code/v/2.1.169, June 2026a. npm package; the system prompt and tool descriptions are shipped with the package. Accessed: 2026-09.

Anthropic. Introducing Claude Sonnet 4.6. https://www.anthropic.com/news/claude-sonnet-4-6, 2026b. Accessed: 2026-08.

Anysphere. Cursor: The AI code editor. https://cursor.com, 2026. Accessed: 2026-08.

Longju Bai, Zhemin Huang, Xingyao Wang, Jiao Sun, Rada Mihalcea, Erik Brynjolfsson, Alex Pentland, and Jiaxin Pei. How do AI agents spend your money? analyzing and predicting token consumption in agentic coding tasks, 2026. URL https://arxiv.org/abs/2604.22750.

Islem Bouzenia and Michael Pradel. Understanding software engineering agents: A study of thought-action-result trajectories. In *2025 IEEE/ACM 40th International Conference on Automated Software Engineering (ASE)*, pp. 2846–2857, 2025. doi: 10.1109/ASE63991.2025.00234. URL https://doi.org/10.1109/ASE63991.2025.00234.

Ira Ceka, Hailie Mitchell, Saurabh Pujar, Luca Buratti, Shyam Ramji, Junfeng Yang, Gail Kaiser, and Baishakhi Ray. Understanding automated program repair agents through the lens of traceability: An empirical study, 2025. URL https://arxiv.org/abs/2506.08311. Accepted at ISSTA 2026.

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. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.acl-long.426. URL https://aclanthology.org/2025.acl-long.426/.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, and Mia Glaese. Introducing SWE-bench verified. https://openai.com/index/introducing-swe-bench-verified/, August 2024. OpenAI Blog. Accessed: 2026-08.

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

Mehmet Hamza Erol, Batu El, Mirac Suzgun, Mert Yuksekgonul, and James Y Zou. Cost-of-pass: An economic framework for evaluating language models. In *International Conference on Learning Representations*, volume 2026, pp. 34103–34127, 2026.

Pengfei Gao and Chao Peng. More with less: An empirical study of turn-control strategies for efficient coding agents, 2025. URL https://arxiv.org/abs/2510.16786.

Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 57–76, 2007. doi: 10.1145/1297027.1297033.

Jingzhi Gong, Ruizhen Gu, Zhiwei Fei, Yazhuo Cao, Lukas Twist, Alina Geiger, Shuo Han, Dominik Sobania, Federica Sarro, and Jie M. Zhang. SkillMOO: Multi-objective optimization of agent skills for software engineering. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE 2026), New Ideas and Emerging Results (NIER) Track*, 2026. doi: 10.1145/3832783.3834550. URL https://arxiv.org/abs/2604.09297. To appear.

Steven N. Goodman. A comment on replication, P-values and evidence. *Statistics in Medicine*, 11 (7):875–879, 1992. doi: 10.1002/sim.4780110705. URL https://doi.org/10.1002/sim.4780110705.

Haoxiang Jia, Earl T. Barr, and Sergey Mechtaev. Compressing code context for LLM-based issue resolution, 2026. URL https://arxiv.org/abs/2603.28119.

Guanyu Jiang, Zhaochen Su, Xiaoye Qu, and Yi R. Fung. XSkill: Continual learning from experience and skills in multimodal agents, 2026. URL https://arxiv.org/abs/2603.12056. Accepted at ICML 2026.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Tomas Kalibera and Richard Jones. Rigorous benchmarking in reasonable time. In *Proceedings of the 2013 International Symposium on Memory Management (ISMM)*, pp. 63–74, 2013. doi: 10.1145/2464157.2464160.

Sayash Kapoor, Benedikt Stroebl, Zachary S. Siegel, Nitya Nadgir, and Arvind Narayanan. AI agents that matter, 2024. URL https://arxiv.org/abs/2407.01502.

Myeongsoo Kim, Dingmin Wang, Siwei Cui, Farima Farmahinifarahani, Terry Yue Zhuo, Shweta Garg, Baishakhi Ray, Rajdeep Mukherjee, and Varun Kumar. Coherence collapse: Diagnosing why code agents fail after reaching the right code, 2026. URL https://arxiv.org/abs/2603.24631.

Wonjoong Kim, Sangwu Park, Yeonjun In, Sein Kim, Dongha Lee, and Chanyoung Park. Beyond the final answer: Evaluating the reasoning trajectories of tool-augmented agents, 2025. URL https://arxiv.org/abs/2510.02837.

LangChain. LangSmith observability. https://docs.langchain.com/langsmith/observability, 2026. Accessed: 2026-08.

Langfuse. Langfuse observability overview. https://langfuse.com/docs/observability/overview, 2026. Accessed: 2026-08.

Xiangyi Li, Wenbo Chen, Yimin Liu, Shenghan Zheng, Xiaokun Chen, Yifeng He, Yubo Li, Bingran You, Haotian Shen, Jiankai Sun, Shuyi Wang, Binxu Li, Qunhong Zeng, Di Wang, Xuandong Zhao, Yuanli Wang, Roey Ben Chaim, Zonglin Di, Yipeng Gao, Junwei He, Yizhuo He, Liqiang Jing, Luyang Kong, Xin Lan, Jiachen Li, Songlin Li, Yijiang Li, Yueqian Lin, Xinyi Liu, Xuanqing Liu, Haoran Lyu, Ze Ma, Bowei Wang, Runhui Wang, Tianyu Wang, Wengao Ye, Yue Zhang, Hanwen Xing, Yiqi Xue, Steven Dillmann, and Han chung Lee. SkillsBench: Benchmarking how well agent skills work across diverse tasks, 2026a. URL https://arxiv.org/abs/2602.12670.

Yu Li, Rui Miao, Zhengling Qi, and Tian Lan. ARISE: Agent reasoning with intrinsic skill evolution in hierarchical reinforcement learning, 2026b. URL https://arxiv.org/abs/2603.16060.

Zheng Li, Qingxiu Dong, Jingyuan Ma, Di Zhang, Kai Jia, and Zhifang Sui. SelfBudgeter: Adaptive token allocation for efficient LLM reasoning. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 21135–21156. Association for Computational Linguistics, 2026c. doi: 10.18653/v1/2026.findings-acl.1063. URL https://aclanthology.org/2026.findings-acl.1063/.

Fulin Lin, Shaowen Chen, Ruishan Fang, Hongwei Wang, and Tao Lin. Stop wasting your tokens: Towards efficient runtime multi-agent systems. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=pzFhtpkabh.

Shuyang Liu, Yang Chen, Rahul Krishna, Saurabh Sinha, Jatin Ganhotra, and Reyhaneh Jabbarvand. Process-centric analysis of agentic software systems. *Proceedings of the ACM on Programming Languages*, 10(OOPSLA1), April 2026a. doi: 10.1145/3798271. URL https://doi.org/10.1145/3798271. Article 163, 28 pages.

Tengxiao Liu, Zifeng Wang, Jin Miao, I-Hung Hsu, Jun Yan, Jiefeng Chen, Rujun Han, Fangyuan Xu, Yanfei Chen, Ke Jiang, Samira Daruki, Yi Liang, William Yang Wang, Tomas Pfister, and Chen-Yu Lee. Budget-aware tool-use enables effective agent scaling, 2025a. URL https://arxiv.org/abs/2511.17006.

Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Fei Wang, Michael Qizhe Shieh, and Wenmeng Zhou. CodexGraph: Bridging large language models and code repositories via code graph databases. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 142–160. Association for Computational Linguistics, 2025b. doi: 10.18653/v1/2025.naacl-long.7. URL https://aclanthology.org/2025.naacl-long.7/.

Xingyan Liu, Xiyue Luo, Linyu Li, Ganghong Huang, Jianfeng Liu, and Honglin Qiao. SkillForge: Forging domain-specific, self-evolving agent skills in cloud technical support. In *Proceedings of the 49th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2026b. doi: 10.1145/3805712.3808466. URL https://doi.org/10.1145/3805712.3808466.

Chiwang Luk, Matin Mohammad Najafi, Zhifeng Jia, Wei Yang, Xiuchang Li, Jinwei Zhu, Yang Ren, Lei Chen, and Gao Cong. ContextSniper: AntTrail's token-efficient code memory for repository-level program repair, 2026. URL https://arxiv.org/abs/2607.01916.

Ziyu Ma, Shidong Yang, Yuxiang Ji, Xucong Wang, Yong Wang, Yiming Hu, Tongwen Huang, and Xiangxiang Chu. SkillClaw: Let skills evolve collectively with agentic evolver, 2026. URL https://arxiv.org/abs/2604.08377.

Oorja Majgaonkar, Zhiwei Fei, Xiang Li, Federica Sarro, and He Ye. Understanding code agent behaviour: An empirical study of success and failure trajectories, 2025. URL https://arxiv.org/abs/2511.00197.

Colby Mchenry. CodeGraph: Semantic code intelligence for AI coding agents. https://github.com/colbymchenry/codegraph, 2026. Accessed: 2026-08.

Evan Miller. Adding error bars to evals: A statistical approach to language model evaluations. *arXiv preprint arXiv:2411.00640*, 2024. URL https://arxiv.org/abs/2411.00640.

MiniMax. MiniMax M3: Frontier coding, 1M context, native multimodality—all in one model. https://www.minimax.io/blog/minimax-m3, 2026. Accessed: 2026-08.

Sania Nayab, Giulio Rossolini, Marco Simoni, Andrea Saracino, Giorgio Buttazzo, Nicolamaria Manes, and Fabrizio Giacomelli. Concise thoughts: Impact of output length on LLM reasoning and cost. *Information Sciences*, 757:123939, 2026. doi: 10.1016/j.ins.2026.123939. URL https://doi.org/10.1016/j.ins.2026.123939.

Jingwei Ni, Yihao Liu, Xinpeng Liu, Yutao Sun, Mengyu Zhou, Pengyu Cheng, Dexin Wang, Erchao Zhao, Xiaoxi Jiang, and Guanjun Jiang. Trace2Skill: Distill trajectory-local lessons into transferable agent skills, 2026. URL https://arxiv.org/abs/2603.25158.

Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E. Gonzalez, M Waleed Kadous, and Ion Stoica. RouteLLM: Learning to route LLMs with preference data, 2024. URL https://arxiv.org/abs/2406.18665.

OpenAI. Introducing codex. https://openai.com/index/introducing-codex/, 2025. Accessed: 2026-08.

Shuyin Ouyang, Jie M. Zhang, Mark Harman, and Meng Wang. An empirical study of the non-determinism of ChatGPT in code generation. *ACM Transactions on Software Engineering and Methodology*, 34(2):1–28, 2025a. doi: 10.1145/3697010. URL https://doi.org/10.1145/3697010. Article 42.

Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. RepoGraph: Enhancing AI software engineering with repository-level code graph. In *International Conference on Learning Representations*, 2025b. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/4a4a3c197deac042461c677219efd36c-Abstract-Conference.html.

Dangfeng Pan, Zhensu Sun, Cenyuan Zhang, David Lo, and Xiaoning Du. The hidden cost of readability: How code formatting silently consumes your LLM budget, 2025. URL https://arxiv.org/abs/2508.13666.

Piebald AI. Claude code system prompts. https://github.com/Piebald-AI/claude-code-system-prompts/tree/06bfbc6, 2026. Prompts extracted from Claude Code v2.1.169. Accessed: 2026-09.

Qwen Team. Qwen3.5: Towards native multimodal agents. https://qwen.ai/blog?id=qwen3.5, 2026. Accessed: 2026-08.

Mohamad Salim, Jasmine Latendresse, SayedHassan Khatoonabadi, and Emad Shihab. Tokenomics: Quantifying where tokens are used in agentic software engineering, 2026. URL https://arxiv.org/abs/2601.14470.

Jun Shao and Shein-Chung Chow. Reproducibility probability in clinical trials. *Statistics in Medicine*, 21(12):1727–1742, 2002. doi: 10.1002/sim.1177. URL https://doi.org/10.1002/sim.1177.

Shivanshu Shekhar, Tanishq Dubey, Koyel Mukherjee, Apoorv Saxena, Atharv Tyagi, and Nishanth Kotla. Towards optimizing the costs of LLM usage, 2024. URL https://arxiv.org/abs/2402.01742.

Hongjin Su, Shizhe Diao, Ximing Lu, Mingjie Liu, Jiacheng Xu, Xin Dong, Yonggan Fu, Peter Belcak, Hanrong Ye, Hongxu Yin, Yi Dong, Evelina Bakhturina, Tao Yu, Yejin Choi, Jan Kautz, and Pavlo Molchanov. ToolOrchestra: Elevating intelligence via efficient model and tool orchestration, 2025. URL https://arxiv.org/abs/2511.21689.

TechCrunch. OpenAI is losing money on its pricey ChatGPT Pro plan, CEO Sam Altman says. https://techcrunch.com/2025/01/05/openai-is-losing-money-on-its-pricey-chatgpt-pro-plan-ceo-sam-altman-says/, January 2025. Accessed: 2026-09.

Fali Wang, Hui Liu, Zhenwei Dai, Jingying Zeng, Zhiwei Zhang, Zongyu Wu, Chen Luo, Zhen Li, Xianfeng Tang, Qi He, and Suhang Wang. AgentTTS: Large language model agent for test-time compute-optimal scaling strategy in complex tasks. In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-3288. URL https://papers.neurips.cc/paper_files/paper/2025/hash/8d9bbba8cac9cabb54e85ee7f21441c8-Abstract-Conference.html.

Yuhang Wang, Yuling Shi, Mo Yang, Rongrui Zhang, Shilin He, Heng Lian, Yuting Chen, Siyu Ye, Kai Cai, and Xiaodong Gu. SWE-Pruner: Self-adaptive context pruning for coding agents, 2026. URL https://arxiv.org/abs/2601.16746.

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

Peng Xia, Jianwen Chen, Xinyu Yang, Haoqin Tu, Jiaqi Liu, Kaiwen Xiong, Siwei Han, Shi Qiu, Haonian Ji, Yuyin Zhou, Zeyu Zheng, Cihang Xie, and Huaxiu Yao. MetaClaw: Just talk – an agent that meta-learns and evolves in the wild, 2026b. URL https://arxiv.org/abs/2603.17187.

Yuan-An Xiao, Pengfei Gao, Chao Peng, and Yingfei Xiong. Reducing cost of LLM agents with trajectory reduction. *Proceedings of the ACM on Software Engineering*, 3(FSE):1241–1263, 2026. doi: 10.1145/3797084. URL https://doi.org/10.1145/3797084.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, 2024. doi: 10.52202/079017-1601. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

Yutao Yang, Junsong Li, Qianjun Pan, Bihao Zhan, Yuxuan Cai, Lin Du, Jie Zhou, Kai Chen, Qin Chen, Xin Li, Bo Zhang, and Liang He. AutoSkill: Experience-driven lifelong learning via skill self-evolution, 2026. URL https://arxiv.org/abs/2603.01145.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Maxwell Zeff. Anthropic unveils new rate limits to curb Claude Code power users. https://techcrunch.com/2025/07/28/anthropic-unveils-new-rate-limits-to-curb-claude-code-power-users/, July 2025. TechCrunch. Accessed: 2026-09.

Guibin Zhang, Junhao Wang, Junjie Chen, Wangchunshu Zhou, Kun Wang, and Shuicheng Yan. AgenTracer: Who is inducing failure in the LLM agentic systems?, 2025. URL https://arxiv.org/abs/2509.03312.

Boyuan Zheng, Michael Y. Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, Gaowen Liu, Graham Neubig, and Yu Su. SkillWeaver: Web agents can self-improve by discovering and honing skills, 2025. URL https://arxiv.org/abs/2504.07079.

Huichi Zhou, Siyuan Guo, Anjie Liu, Zhongwei Yu, Ziqin Gong, Bowen Zhao, Zhixun Chen, Menglong Zhang, Yihang Chen, Jinsong Li, Runyu Yang, Qiangbin Liu, Xinlei Yu, Jianmin Zhou, Na Wang, Chunyang Sun, and Jun Wang. Memento-Skills: Let agents design agents, 2026. URL https://arxiv.org/abs/2603.18743.

Table 5: Action-label taxonomy.

Category	Label	Description
Retrieval	`Retrieve_Structure`	Searches or explores the repository structure or current location.
	`Retrieve_Code`	Reads or searches source code files, except test files or documents.
	`Retrieve_Tests`	Reads or searches test files specifically.
	`Retrieve_Docs`	Reads documentation (non-code references such as README).
Modification	`Generate_Patch`	Modifies a file inside the working tree (e.g., `/testbed`) to fix the issue.
	`Generate_Tests`	Writes tests or reproduction scripts (e.g., `python -c`).
	`Generate_Tools`	Creates custom tools for the agent's own use.
	`Create_Files_Directories`	Creates new files or directories (e.g., `touch`, `mkdir`).
	`Remove_Content`	Removes code or files the agent previously created.
Verification	`Execute_Existing_Tests`	Runs existing repository tests.
	`Execute_Generated_Tests`	Executes agent-generated code.
	`Review_Patch`	Reviews the applied patch (e.g., `git diff`).
Git	`Execute_Git`	Runs git commands other than `git diff`.
Other	`Finish_Task`	Submits the final solution.
	`Env_Inspection`	Checks or installs packages; inspects environment variables or resources.
	`Pure_Reasoning`	The step contains only reasoning, without calling any tool.
	`Delegate_Subagent`	Dispatches a subagent via the `Agent` tool (*CC* only).
	`Other`	Does not fit any label above.

A Appendix

This appendix provides supplementary details for our study. First, we describe additional study setup details, including the versions and configurations of Claude Code and Mini-SWE-Agent, action-labeling procedures, benchmark task split, and how we handle the agent execution randomness. Second, we provide additional results and analysis for RQ1, including the performance of the four agent configurations on the 300-task RQ1 analysis set, extra detection details, analysis, and examples of the three cost-inefficient behaviors. Third, we provide CodeGraph integration details and further results analysis for RQ2. Finally, we detail the implementation of agent-synthesized skills and provide the full content of both agent-synthesized and developer-designed skills in RQ3.

A.1 Additional Details for Study Setup

In this section, we will provide the specific version and configuration details of the agents included in our study, and then we will introduce the agent action labeling process, benchmark task split, and the details of handling agent execution randomness.

We use the same agent versions and configurations across all RQs. *CC* uses CLI v2.1.169 (Anthropic, 2026a) with a $3.00 per-task cost limit for SWE-bench Verified tasks and a $5.00 per-task cost limit for SWE-bench Pro tasks; the backbone model temperature is not exposed through the CLI. *MSA* uses v2.3.0 with temperature 0.0, a 250-step limit, and the same per-task cost limit as CC. We next elaborate on the action-labeling procedure, benchmark split details, and our treatment of execution randomness described in Section 2.

A.1.1 Action Labeling

Why label actions. The action labels serve four purposes. First, the two agent frameworks expose actions in different surface forms: *CC* issues named tool calls while *MSA* issues raw shell commands. We therefore assign every step one or two semantic action labels before any detection, which projects both frameworks onto a shared vocabulary and makes their trajectories comparable. Second, they restrict each detector to its relevant step population: SubRetrv compares returned context only across retrieval steps, SimScrpt mines scripts only from script-generation or script-execution steps, and ReTest groups only test-execution steps. Third, they provide the structural boundaries the detections rely on. For example, a step labeled `Generate_Patch` closes the current ReTest group, so re-running a test after an updated patch is never flagged. Finally, they support the aggregate analysis, such as the retrieval-action accounting in Table 2 (Patch-Adjacent SubRetrv and Within-Sequence SubRetrv). The same labeling pipeline is applied uniformly to every run in RQ1–RQ3.

Label taxonomy. We use 18 labels in five categories, summarized in Table 5. Action `Delegate_Subagent` arises only in *CC*'s tool set. A tool call may carry up to two labels. For example, an inline `python -c` reproduction script both authors and executes agent-generated code (`Generate_Tests` & `Execute_Generated_Tests`).

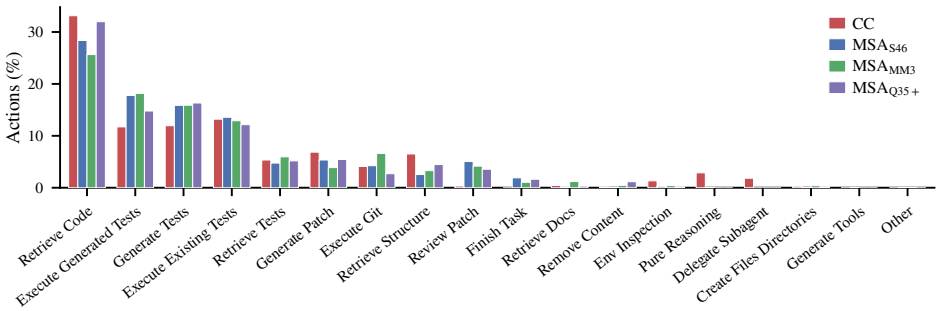


Figure 7: Distribution of action labels over the RQ1 trajectories.

Table 6: Task selection by repository on SWE-bench Verified and SWE-bench Pro.

SWE-bench Verified (500 tasks)			**SWE-bench Pro** (731 public tasks)		
Repository	**Analysis**	**Eval.**	**Repository**	**Pool**	**Eval.**
astropy/astropy	13	9	ansible/ansible	96	12
django/django	139	92	element-hq/element-web	56	8
matplotlib/matplotlib	20	14	flipt-io/flipt	85	10
mwaskom/seaborn	1	1	future-architect/vuls	62	9
pallets/flask	1	0	gravitational/teleport	76	10
psf/requests	5	3	internetarchive/openlibrary	91	11
pydata/xarray	13	9	navidrome/navidrome	57	8
pylint-dev/pylint	6	4	NodeBB/NodeBB	44	8
pytest-dev/pytest	12	7	protonmail/webclients	65	9
scikit-learn/scikit-learn	19	13	qutebrowser/qutebrowser	79	10
sphinx-doc/sphinx	26	18	tutao/tutanota	20	5
sympy/sympy	45	30			
Total	**300**	**200**	**Total**	**731**	**100**

Rule-based labeling. Steps are first labeled by deterministic rules, with one rule family per harness. For *MSA*, whose actions are raw shell commands, each command is split into atomic sub-commands (`&&`, `;`, and pipes) after unwrapping prefixes such as `timeout`; navigation-only fragments (`cd`, `export`) are dropped. Read-only verbs (`cat`, `head`, `sed -n`, `grep`, `ls`, `find`, ...) map to *Retrieval*, with test-path, documentation-path, and source-extension patterns selecting the sub-label. Write channels (heredocs, `sed -i`, `tee`, output redirection) map to *Modification*. Modifying an existing file inside the working directory yields `Generate_Patch`; a test-looking file yields `Generate_Tests`; ephemeral script generations (`python -c`, `python <<EOF`) are labeled as `Generate_Tests` + `Execute_Generated_Tests`, unless they call an existing test API or open a working-tree file for writing. Test runners map to `Execute_Existing_Tests` or `Execute_Generated_Tests` depending on their target; `git diff` maps to `Review_Patch`; other `git` commands map to `Execute_Git`; the submission marker (MSA only) maps to `Finish_Task` ; and steps without any command map to `Pure_Reasoning`. For *CC*, whose actions are named tool calls, tools map directly (`Read`/`Grep` to *Retrieval* by path, `Glob` to `Retrieve_Structure`, `Edit`/`Write` to *Modification* by path, `Agent` to `Delegate_Subagent`), and `Bash` calls are routed through the same command classifier as *MSA*.

LLM fallback. A step is forwarded to an LLM judge (GPT-5.1) whenever any of its atomic commands matches no rule. The judge receives the step's thought and commands together with the taxonomy definitions and returns at most two labels with a brief justification. Among the RQ1 trajectories, the fallback handles 0.94% of *CC* steps, 31.17% of MSA_{S46} steps, 1.62% of MSA_{MM3} steps, and 1.18% of MSA_{Q35+} steps, accounting for 7.26% of the total steps of four agent configurations on 300 Verified tasks. By submitting all LLM-labeling requests through the batch mode, the labeling cost is less than 0.2 cents per trajectory.

Resulting distribution. Figure 7 shows the label distribution over the RQ1 trajectories among four agent configurations. The pattern is consistent across configurations: reading source code (`Retrieve_Code`, 25.65–33.12%) together with generating and executing scripts (`Generate_Tests` + `Execute_Generated_Tests`, 23.66–34.07%) dominate every configuration, followed by existing-test execution (12.15–13.57%). Architecture differences appear in the tail: only *CC* delegates work to subagents (`Delegate_Subagent`, 1.83% of its actions), while *MSA* configurations review patches through `git diff` more often (3.57–5.04% vs. 0.32%).

Table 7: Noise floor of the eight baselines. The floor s is the standard deviation of Pass@1 (pp) and the coefficient of variation of cost and CoP (%). The last two columns give the smallest single-run $|\Delta|$ classified as robust ($1.90\,s$). Per-run cost is averaged over the tasks completed in all three runs.

		Pass@1		Cost		CoP	Min. robust $\lvert\Delta\rvert$, $R{=}1$	
Benchmark	**Config**	Run 1 / 2 / 3 (%)	s_{pass} (pp)	Run 1 / 2 / 3 ($)	s_{cost} (%)	s_{CoP} (%)	Cost	Pass@1 (pp)
Verified-200	CC	74.00 / 75.50 / 75.50	0.87	0.598 / 0.525 / 0.515	8.35	9.56	15.87%	1.65
Verified-200	MSA_{S46}	72.00 / 71.50 / 74.00	1.32	0.674 / 0.605 / 0.636	5.40	5.49	10.25%	2.51
Verified-200	MSA_{MM3}	78.00 / 71.50 / 76.00	3.33	0.463 / 0.442 / 0.477	3.80	2.85	7.21%	6.33
Verified-200	$\text{MSA}_{\text{Q35+}}$	65.00 / 70.50 / 66.00	2.93	0.123 / 0.110 / 0.115	5.70	9.68	10.82%	5.57
Pro-100	CC	55.00 / 56.00 / 52.00	2.08	0.917 / 0.922 / 0.881	2.44	1.94	4.64%	3.95
Pro-100	MSA_{S46}	55.00 / 49.00 / 52.00	3.00	1.037 / 1.067 / 0.983	4.14	8.43	7.87%	5.70
Pro-100	MSA_{MM3}	60.00 / 64.00 / 56.00	4.00	0.859 / 0.852 / 0.896	2.70	9.32	5.13%	7.60
Pro-100	$\text{MSA}_{\text{Q35+}}$	47.00 / 41.00 / 47.00	3.46	0.126 / 0.126 / 0.132	2.83	7.22	5.38%	6.58

A.1.2 Behavior Cost Attribution

For each action flagged as cost-inefficient, we use the `o200k_base` tokenizer from `tiktoken` to count the tokens in its reasoning, commands, and tool output. We then attribute costs using the corresponding model's token prices. Generated content is charged at the output-token price, while both generated content and tool output incur input-token or cache-token charges when carried into subsequent LLM requests. We sum these charges to obtain the action's attributed cost.

A.1.3 Benchmarks & Task Split

Table 6 lists the tasks per repository used for analysis in **RQ1** and for evaluation in **RQ2–RQ3**. For *SWE-bench Verified*, we order the tasks of each repository by creation date and assign the earliest tasks, about 60% per repository, to the analysis set and the remaining ones to *Verified-200*. `pallets/flask` has a single task, which falls into the analysis set.

For *SWE-bench Pro*, we draw a stratified random sample of 100 tasks from the 731 public tasks with a fixed seed. The *Pool* column gives the public tasks available in each repository. Each repository receives a quota that grows with the square root of its pool size, with a minimum of four tasks, so that all 11 repositories are represented, and repositories with more tasks do not crowd out smaller ones. Within each repository, tasks are ranked by their cost in the public SWE-bench Pro leaderboard run of SWE-agent (Yang et al., 2024) with Claude Sonnet 4.5 (250-turn limit, no cost cap)[4] and split into three equal-sized terciles. The low, mid, and high terciles are sampled at 20%, 30%, and 50%, since costlier tasks are expected to expose more cost-inefficient behaviors.

A.1.4 Controlling Agent Execution Randomness

This section describes how the run-to-run noise floors in **RQ2–RQ3** are measured, how the effect of single-run approach cells is judged, and how the underlying equal-variance assumption was validated. All numbers are produced by the same pipeline that generates Table 3.

Design Overview. Re-running every (benchmark, configuration, approach, and baseline) cell three times ($2 \times 4 \times (3+1) \times 3 = 96$ runs in total) would cost about $7,500 at the per-run costs we observed, so we proceed in three steps with selective re-running 8 baseline settings and 9 approach cells. First, every *baseline* setting (two benchmarks $\times$ four agent configurations) is executed three times to estimate the run-to-run noise floors in Table 7, following standard practice in performance benchmarking (Georges et al., 2007; Kalibera & Jones, 2013) and LLM evaluations (Miller, 2024). Second, 15 approach cells are executed once, and their effects are judged against the noise floors transferred from their corresponding baseline, *under the equal-variance assumption that our approaches do not materially alter the sampling and infrastructure nondeterminism driving run-to-run noise*. We further verify this assumption directly on the nine replicated cells in Table 8, selected to cover both benchmarks, all three approaches, and both agent frameworks at low replication cost. Here are the nine approach cells we select to replicate: six cells run all three approaches on both benchmarks with $\textit{MSA}_{\textit{Q35+}}$, the cheapest configuration to replicate; one cell runs developer-designed skills on Verified-200 with $\textit{MSA}_{\textit{MM3}}$, the noisiest configuration based on its baseline noise

[4] When we selected the tasks on June 26, 2026, this was the best-performing leaderboard run whose released trajectories include cost information (43.70% resolved). The share of resolved tasks in *Pro-100* matches that of this run, with 44 passed tasks.

Table 8: Approach cells executed three times to verify the equal-variance assumption. Each metric lists the approach and baseline noise (s_{appr} / s_{base}) and their ratio. The remaining 15 approach cells are single runs.

			Pass@1 (pp)			Cost ($)			CoP
Benchmark	**Config**	**Approach**	Per run	s_{appr} / s_{base}	Ratio	Per run	s_{appr} / s_{base}	Ratio	Ratio
Verified-200	CC	DevSkills	73.00 / 73.50 / 74.00	0.50 / 0.87	0.58	0.497 / 0.487 / 0.435	7.06 / 8.35	0.84	0.80
Verified-200	CC	CodeGraph	73.50 / 73.50 / 75.50	1.15 / 0.87	1.33	0.590 / 0.613 / 0.587	2.36 / 8.35	0.28	0.36
Verified-200	MSA_{MM3}	DevSkills	72.00 / 72.50 / 74.50	1.32 / 3.33	0.40	0.371 / 0.381 / 0.410	5.23 / 3.80	1.38	1.19
Verified-200	$\text{MSA}_{\text{Q35+}}$	DevSkills	67.50 / 72.00 / 70.00	2.25 / 2.93	0.77	0.107 / 0.099 / 0.103	3.83 / 5.70	0.67	0.73
Verified-200	$\text{MSA}_{\text{Q35+}}$	SynSkills	70.00 / 66.00 / 73.00	3.51 / 2.93	1.20	0.119 / 0.112 / 0.115	3.01 / 5.70	0.53	0.48
Verified-200	$\text{MSA}_{\text{Q35+}}$	CodeGraph	73.00 / 72.00 / 72.50	0.50 / 2.93	0.17	0.109 / 0.115 / 0.113	3.01 / 5.70	0.53	0.38
Pro-100	$\text{MSA}_{\text{Q35+}}$	DevSkills	48.00 / 42.00 / 48.00	3.46 / 3.46	1.00	0.140 / 0.139 / 0.142	1.38 / 2.83	0.49	0.97
Pro-100	$\text{MSA}_{\text{Q35+}}$	SynSkills	49.00 / 45.00 / 44.00	2.65 / 3.46	0.76	0.130 / 0.132 / 0.133	1.33 / 2.83	0.47	1.03
Pro-100	$\text{MSA}_{\text{Q35+}}$	CodeGraph	41.38 / 35.63 / 40.23	3.04 / 4.14	0.73	0.146 / 0.164 / 0.141	8.19 / 3.43	2.39	1.51

floor; the remaining two cells, whose single-run effects are borderline, run developer-designed skills and CodeGraph on Verified-200 with *CC*. In total, the **RQ2–RQ3** evaluation comprises over 10k trajectories in SWE-bench Verified and Pro. For RQ1, each agent configuration is only executed once on 300 Verified tasks. The remainder of this section details these steps in turn: the measured noise floors, the decision rule that classifies an observed effect as robust, the validation of the equal-variance assumption together with a stress test showing that the verdicts hold even if it were violated, and the reason behavior metrics carry no robustness verdicts.

Noise floors. Table 7 lists the three baseline runs of eight settings. The noise floor s is the standard deviation (SD) of Pass@1 (in pp), and the coefficients of variation (CV) of cost (in %) and CoP (in %). Pass@1 floors range from 0.87 pp (*CC* on Verified-200) to 4.00 pp (MSA_{MM3} on Pro-100). The stability order within each benchmark is $CC > MSA_{S46} > MSA_{Q35+} > MSA_{MM3}$. Cost floors range from 2.44% to 8.35%. CoP floors are generally wider than cost floors (up to 3.45×, MSA_{MM3} on Pro-100) because CoP inherits the variance of the Pass@1 in its denominator. For CodeGraph on Pro-100, which runs on 87 tasks (Appendix A.3.1), all noise floors are re-estimated on the same 87 tasks: 3.51, 3.51, 3.45, and 4.14 pp for Pass@1, 2.59%, 3.99%, 2.67%, and 3.43% for cost, and 5.91%, 9.20%, 8.28%, and 10.81% for CoP, for *CC*, MSA_{S46}, MSA_{MM3}, and MSA_{Q35+}, respectively.

Robust decision rule. Cost and CoP changes (Δ) are computed on the tasks completed in every run of both the approach and its baseline. For an approach cell with R runs compared to its three-run baseline, the standard error of Δ is $\sigma_\Delta = s\sqrt{1/R + 1/3}$. We classify Δ as robust when $|\Delta|/\sigma_\Delta \geq 1.645$, i.e., when a replication would reproduce its sign with probability at least 95% under a normal model (Goodman, 1992; Shao & Chow, 2002). **For a single-run cell this reduces to $|\Delta| \geq 1.90\, s$, which is a stricter bar than a replicated cell's**. The resulting thresholds for each setting are given in the last two columns of Table 7.

Validating the equal-variance assumption. Table 8 reports nine approach cells executed three times. All other 15 approach cells are single runs. Across the nine replicated cells, the approach-side Pass@1 SD is 0.17–1.33× the baseline's, and the cost SD is 0.28–1.38× the baseline's in eight of the nine cells. However, these ratios are themselves imprecise: for three normal runs, the sample variance satisfies $2s^2/\sigma^2 \sim \chi^2_2$, so an estimated s carries a relative standard error of $\sqrt{1-\pi/4}\,/\,(\sqrt{\pi}/2) \approx 52\%$, and ratios moderately above one are expected even under equal noise. Approach runs are thus not systematically noisier than baseline runs, which is what the transfer of baseline noise floors requires. The single exception is CodeGraph with MSA_{Q35+} on Pro-100, whose cost noise is 2.39× its baseline's (8.19% vs. 3.43%). For this cell we use its own observed noise floor (8.19% for cost and 16.34% for CoP). Its cost and CoP increases remain robust. The same amplification is not observed for CodeGraph with *CC* on Verified-200 (cost ratio 0.28), so we do not widen CodeGraph floors elsewhere.

Moreover, the verdicts in Table 3 carry a safety margin beyond this validation: even if the equal-variance assumption were violated at the largest magnitude we observed outside the exception, none of these verdicts would change for two reasons. First, every single-run effect classified as robust remains robust even after its transferred noise floor is inflated by 1.5×, whereas the largest amplification observed among the replicated cells outside the exception is only 1.38× (the cost of DevSkills with MSA_{MM3} on Verified-200, in Table 8). Second, though each robust effect retains a residual probability of reversing its sign in a replication, summing these probabilities over the 13 robust cost effects yields only 0.06 expected sign reversals if the entire study were rerun.

Behavior metrics. The per-task behavior frequencies reported in Table 3 are considerably noisier than cost. Across the eight baselines, the CV of a behavior's per-task frequency over the three runs averages 15.68% and reaches 44.47% (SimScrpt of MSA_{Q35+} on Pro-100), whereas cost floors range from 2.44% to 8.35%; it exceeds the cost floor of the same setting in 20 of the 24 baseline-behavior combinations. We therefore report behavior changes in Table 3 as mechanistic evidence and do not attach robustness verdicts to them.

A.2 Additional Results for RQ1

In this section, we provide additional results and analysis for RQ1. We first report Pass@1, task cost, and CoP for the four agent configurations on the 300 RQ1 tasks. We further provide additional statistics, analysis, and illustrative examples for the three identified cost-inefficient behaviors.

Task performance and cost. Table 9 reports Pass@1, average task cost, and CoP of the four agent configurations on the RQ1 analysis set. *CC* achieves the highest Pass@1 (76.67%) at a per-task cost slightly below MSA_{S46} ($0.52 vs. $0.55). MSA_{MM3} trades 1.67 pp of Pass@1 against MSA_{S46} for a 34.04% lower task cost, and MSA_{Q35+} is the cheapest configuration ($0.10 per task) but also the least accurate (65.67%).

Table 9: Pass@1, cost, and CoP of the four agent configurations in **RQ1**.

Config	Pass@1	Cost ($)	CoP
CC	76.67%	0.520	0.679
MSA_{S46}	74.33%	0.554	0.745
MSA_{MM3}	72.67%	0.365	0.502
MSA_{Q35+}	65.67%	0.102	0.156

A.2.1 Why CC Exhibits Fewer Cost-Inefficient Behaviors than MSA

A large part of *CC*'s advantage lies in its built-in system prompt and tool descriptions (Anthropic, 2026a; Piebald AI, 2026). They already contain instructions that mitigate inefficient behaviors before any external optimization approach is applied, as listed below.

SubRetrv:

- `Bash` tool description: "IMPORTANT: Avoid using this tool to run commands, unless ... a dedicated tool cannot accomplish your task. Instead, use the appropriate dedicated tool...", with an explicit mapping: file search via `Glob` instead of `find`, content search via `Grep` instead of `grep`, file reading via `Read` instead of `cat`.
- `Agent` tool (used for delegating subagents) description: "Launch up to N agents IN PARALLEL (single message, multiple tool calls) to efficiently explore the codebase. Use 1 agent when the task is isolated to known files..." However, this is double-edged: it keeps the main context compact, but the summaries omit the exact code, so the main agent sometimes re-reads the same region later. This causes the Cross-Agent SubRetrv that accounts for 50.15% of *CC*'s SubRetrv (Table 2).

However, no built-in instructions contain context like "do not re-read what you already retrieved", which is one of the reasons why *CC* still has SubRetrv, and other mitigation approaches can still optimize such behavior.

SimScrpt:

- System prompt: "NEVER create files unless they're absolutely necessary for achieving your goal. ALWAYS prefer editing an existing file to creating a new one." It suppresses scratch file creation at the source, making file-based SimScrpt rare on *CC*. *CC* creates 0.21 draft files per task versus 2.72 to 7.07 for *MSA*, and file-based SimScrpt appears in only two *CC* tasks.
- `Edit` tool description: "ALWAYS prefer editing existing files. NEVER write new files unless explicitly required." The rich feedback of the `Edit` tool and such descriptions make CC barely generate scripts for code editing, which is a major category in *MSA*'s file-based SimScrpt.

Since creating an issue reproduction script can still help solve the task, *CC* achieves this with inline `python -c` one-shots. This explains why its SimScrpt is almost entirely ephemeral rather than file-based.

ReTest: we find no built-in instruction that discourages re-running tests without updated patches or asks the agent to reuse previous test output. Consistently, ReTest is the one behavior where *CC* shows no clear advantage in frequency: 2.09 executions per task, the same magnitude as MSA_{S46} (2.51) and MSA_{Q35+} (2.10).

Other instructions improve cost efficiency but are not tied to the specific cost-inefficient behaviors we identified. "You have the capability to call multiple tools in a single response. When multiple independent pieces of information are requested and all commands are likely to succeed, run multiple tool calls in parallel for optimal performance." helps reduce round trips; "Your responses should be short and concise." reduces output tokens.

A.2.2 SUBSUMED RETRIEVAL

Our detection mechanisms for SubRetrv do not consider code clones beyond whitespace differences, which may underestimate SubRetrv; we leave clone-aware detection to future work.

Table 10 quantifies the retrieved context behind SubRetrv: across configurations, agents re-read 11.05–18.10% of all retrieved code lines through SubRetrv actions. Notably, retrieval volume alone does not determine SubRetrv severity. MSA_{S46} reads the fewest lines per task (573) yet re-reads a larger share (16.69%) than *CC*, which reads almost twice as many (1,032) with the lowest share (11.05%). How the architecture manages the retrieved context therefore matters more than how much it retrieves. Below, we illustrate SubRetrv occurrence scenarios with concrete examples.

Table 10: Total retrieved context and the lines re-read by SubRetrv actions.

Metric	CC	MSA_{S46}	MSA_{MM3}	MSA_{Q35+}
Avg. total lines read	1,032	573	912	1,051
Avg. lines re-read by SubRetrv	114	96	137	190
Overlapping rate	11.05%	16.69%	15.00%	18.10%

Patch-Adjacent SubRetrv. As illustrated in Figure 8, on `django-13158`, MSA_{Q35+} attempts to modify `set_empty` at step 40 using `sed -i`, but the edit silently fails. The agent then re-reads the target region at step 41, detects that the function remains unchanged, and subsequently generates a Python script to apply the patch successfully. However, the step-41 retrieval is fully covered by an earlier retrieval at step 29, making it a SubRetrv instance. This example illustrates how limited and error-prone editing support can induce Patch-Adjacent SubRetrv: when an editing tool is difficult to use reliably or provides insufficient feedback about the resulting code state, the agent may re-read previously retrieved code to verify or recover from the edit.

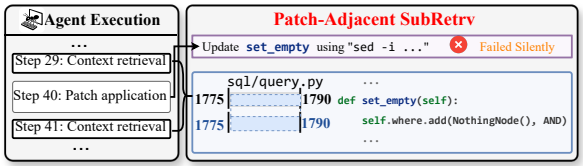


Figure 8: Patch-Adjacent SubRetrv of MSA_{Q35+} on SWE-bench Verified task `django-13158`.

Within-Sequence SubRetrv. As illustrated in Figure 9, on `astropy-8707`, MSA_{Q35+} reads the entire `fits/header.py` (2,243 lines) with `cat` at step 2. The agent then zooms back into the same file for specific functions and precise localization during the following retrieval sequence: steps 3, 10, and 11 re-read 74, 79, and 47 lines respectively, each fully contained in the step-2 read.

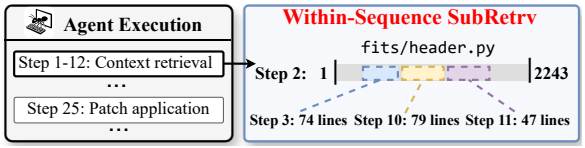


Figure 9: Within-Sequence SubRetrv of MSA_{Q35+} on SWE-bench Verified task `astropy-8707`.

Long-Distance SubRetrv. As illustrated in Figure 10, on `matplotlib-24627`, MSA_{MM3} reads lines 150–200 of `artist.py` at step 22 and re-retrieves lines 184–196 of the same file at step 194, after a 172-step gap. The re-read is triggered by a long debugging loop and a progress stall: at steps 184–193 the agent runs the same pickle test six times while its reasoning repeatedly reverses itself ("Wait, let me check…", "Actually, let me check…", "Hmm, the test…"). To re-anchor its analysis, the agent re-reads the previously retrieved region, along with regions of `test_pickle.py` and `_version.py` that it had also retrieved before.

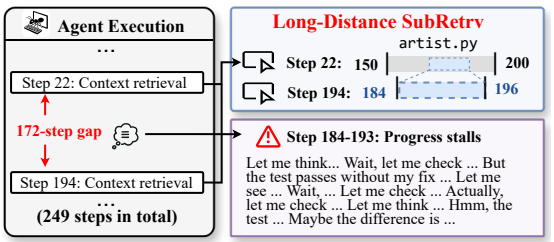


Figure 10: Long-Distance SubRetrv of MSA_{MM3} on SWE-bench Verified task `matplotlib-24627`.

A.2.3 SIMILAR SCRIPT GENERATION

Classifying script functionalities. We classify each SimScrpt script by its function using AST analysis. A script is a *patch script* if it opens a file for writing and transforms its content, through `re.sub`, `str.replace`, or a read-modify-write sequence. A script is classified as an *assertion test* if it contains `assert` statements, imports `pytest` or `unittest`, or calls `self.assert*` methods. A script is an *inspection probe* if it only queries or prints runtime information, such as `print`, `repr`, and `dir` calls, without producing a pass or fail verdict. Scripts matching several signals are assigned by priority: patch, then test, then probe. Scripts that cannot be parsed by AST (0.61–5.54%) are classified using pattern matching.

Ephemeral SimScrpt. The motivating example in Figure 1 already contains this behavior. To reproduce the `union().none()` issue on `django-13158`, *CC* emits four inline `python -c` scripts of 23 to 29 lines. Every script re-emits the same logic: ① configuring Django settings, ② initializing the test environment and database, ③ creating test data, ④ building the querysets, and ⑤ verifying the `.none()` behavior that the issue asks to fix. Yet the four scripts differ by only a few lines each: after a missing-table failure in earlier probes, the first script updates the database initialization with `migrate --run-syncdb`; the second updates the test-environment setup; the third expands the combined-query test scenarios, replacing `count()` with `list()` after Django rejects `count()` on combined querysets; and the fourth clears the query ordering with `order_by('')` after a default-ordering conflict in the union query. Because each script vanishes after execution, the agent regenerates the identical script logic on every iteration.

File-based SimScrpt. As illustrated in Figure 11, on `matplotlib-24627`, MSA_{MM3} writes `/tmp/patch.py` at step 55 to modify `axes/_base.py`, then creates `/tmp/patch2.py` at step 93 and `/tmp/patch3.py` at step 113 as its fix evolves. The three scripts share the same scaffold, which reads the file, applies a string replacement, and writes the file back; only the strings for replacement change. Although the script already exists on disk, the agent versions its fix by writing a near-duplicate sibling file each time instead of editing the existing one.

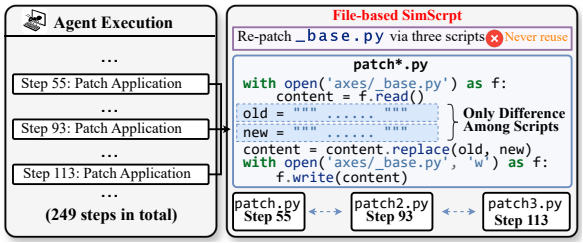


Figure 11: File-based SimScrpt of MSA_{MM3} on SWE-bench Verified task `matplotlib-24627`.

Ideally, to prevent the File-Based SimScrpt in Figure 11, the agent would persist a reusable `patch.py` that accepts the target file and the old and new strings as parameters. The script should provide explicit feedback on whether the replacement succeeded and report specific errors, such as missing or ambiguous matches. The agent could then reuse the same script for subsequent patches, avoiding repeated script generation and its associated cost.

A.2.4 TEST RE-EXECUTION

Identifying re-executions. The executed tests (test ID) are extracted from the commands labeled as `Execute_Existing_Tests` and `Execute_Generated_Tests` from each inter-patch window. For example, the following commands `pytest tests/foo/test_bar.py`, `python -m django test foo.test_bar`, and `python tests/runtests.py foo.test_bar` map to the same test ID: `test_bar`. Flags and output filters such as `-v` or `tail` do not affect the extracted tests, and the test IDs of the agent-generated test scripts are recorded by their file path. Within each group with the same test ID, we conservatively retain the final execution as potentially decision-relevant and flag all preceding executions as ReTest.

Figure 12 shows that ReTest frequency is long-tailed. Depending on the configuration, 23 to 107 affected tasks contain at least six ReTest actions. The tail is most pronounced for MSA_{MM3}, where 107 of the 249 affected tasks contain at least six ReTest actions and 50 contain at least ten, compared with 39 and 8 of 149 for *CC*. Figure 13 shows that 48.01–72.84% of ReTest actions occur at a step distance of one, immediately following the previous execution. *CC* is the most concentrated, with 72.84% at distance one and only 1.76% at ten or more, whereas MSA_{MM3} has the heaviest long-distance tail at 18.78%. Immediate ReTest is consistent with test knowledge gaps and test-signal

recovery, where the agent retries with a different runner or output filter, while long-distance ReTest is consistent with progress stalls. Below, we illustrate the three causes with concrete examples.

Repository-specific test knowledge gaps. Agents may lack knowledge of a repository's expected test runner, working directory, or configuration, leading to repeated failed validations. For example, in Figure 1, *CC* makes eight failed attempts to run `test_qs_combinators`: it repeatedly uses `python -m django test`, which cannot resolve the `queries` module. *CC* then inspects Django's test structure and discovers its repository-specific test runner. It then invokes `tests/runtests.py` with the correct test label, successfully passing all 31 tests.

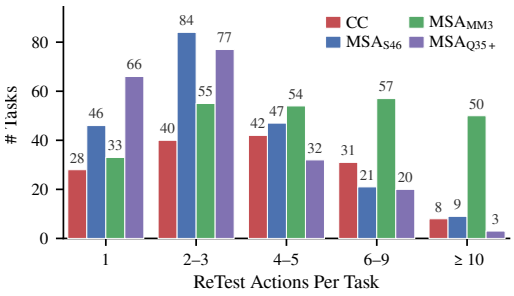


Figure 12: Distribution of ReTest actions per task.

Test-signal recovery. A test execution may fail to provide usable signals because its output is truncated, ambiguous, or not effectively captured by the agent, prompting repeated execution to recover the missing information. Such reruns can be locally reasonable because understanding the failure signal is critical for deciding the next debugging step, but they reveal inefficiency in how test results are delivered and retained. For example, on `sympy-13878` (Figure 14), *MSA$_{MM3}$* runs `test_gamma` eight times between steps 131–139 while applying different test feedback filters and keyword searches to diagnose two reported exceptions. The failures stem from a Python 3.9 compatibility issue that promotes a `DeprecationWarning` to an exception, but *MSA$_{MM3}$* fails to capture an explicit exception marker from the filtered output such as `Error`, `Traceback`, or `Exception`. As a result, the agent repeatedly greps and truncates the test output without isolating the cause. It eventually uses `git stash` and reruns the test, successfully identifying the environment-related issue.

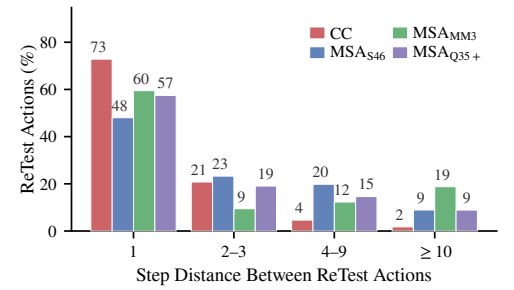


Figure 13: Distribution of step distances between ReTest actions within a ReTest group.

Progress stalls. In some cases, prior executions already provide clear test signals, yet the agent continues running the same tests without modifying the patch. In these cases, the agent often exhibits decision-making stalls, repeating the same reasoning and validation without advancing the solution. For example, on `django-11141`, *MSA$_{S46}$* applies a patch at step 66, with only `test_load_empty_dir` failing because it encodes the legacy behavior that the issue intends to remove. The agent recognizes that it should either update this outdated test, which is the correct solution, or redesign its implementation to preserve the old behavior. However, *MSA$_{S46}$* hesitates over whether updating the test functions satisfies the instruction requirements. It consequently reruns the `migrations` test suite 22 times and `test_makemigrations_no_init` 18 times without another patch until the step budget is exhausted and the task fails.

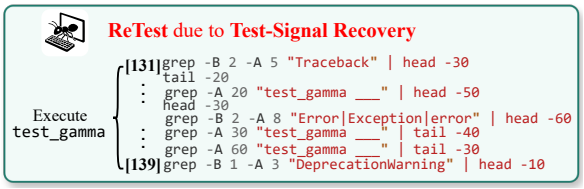


Figure 14: Test-signal recovery of *MSA*$_{\text{MM3}}$ on SWE-bench Verified task `sympy-13878`.

A.3 Additional Details for RQ2

This section provides the additional details for Section 4.2. Appendix A.3.1 describes why we choose CodeGraph, and how CodeGraph is integrated into the two agent frameworks. Appendix A.3.2 reports how agents actually use CodeGraph (Table 12).

A.3.1 CodeGraph Integration

Why CodeGraph. We choose CodeGraph as a representative tool of structure-aware retrieval for two reasons. First, it is one of the most widely adopted tools of this kind: when we ran our exper-

iments in July 2026, the open-source repository had 63.8K stars on GitHub, and its npm package was downloaded 359K times in that month. Second, it robustly supports integration into both agent frameworks: it ships an MCP server for *CC* and a CLI for *MSA*, and it covers 22 languages in the version we use, including Python, JavaScript, TypeScript, and Go, the languages of the SWE-bench Verified and Pro tasks.

We use CodeGraph v0.9.9 (Mchenry, 2026), an open-source code-intelligence tool that parses a repository into an AST-based index of symbols and their relations (definitions, calls, inheritance, and test coverage). The index is built once per task before the agent starts and is refreshed automatically when the agent edits files. CodeGraph is read-only; it never modifies files or runs code. *CC* accesses it through an MCP server that exposes seven functionalities, and *MSA* accesses it through two types of shell commands: `cgsrc` (read a symbol's source code by name) and `cg` (locate symbols, trace call relationships, estimate change impact, list files). Both harnesses receive equivalent CodeGraph instructions in the system and task prompts. The instructions designate CodeGraph as the primary way to locate and read symbols and to trace call relationships, instead of using `grep` or `find` for symbol lookup. Ordinary shell commands and tools remain allowed for non-symbol text retrieval such as configuration files, documentation, and error-message strings, and for editing, execution, and other purposes.

Task coverage on Pro-100. CodeGraph cannot run on 13 of the Pro-100 tasks. For 11 of them, the task image is built on Alpine Linux, which uses the musl C library, whereas CodeGraph ships a Node.js binary compiled against glibc, so the binary cannot be executed in these containers. CodeGraph also fails on the remaining two tasks, one from `ansible` and one from `teleport`. We have not identified the specific cause. We therefore evaluate CodeGraph on the other 87 tasks and compare it with the baseline restricted to the same 87 tasks. On this subset, the baseline Pass@1 is 53.26%, 49.81%, 58.62%, and 42.53% for *CC*, MSA_{S46}, MSA_{MM3}, and MSA_{Q35+}, respectively.

Table 11: CodeGraph interfaces of *CC* (MCP tools) and *MSA* (shell commands), using function `json_script` in `django/utils/html.py` as an example.

Functionality	***CC* tool call**	***MSA* command**
Read source	`codegraph_explore(query="json_script")`	`cgsrc "json_script"`
Locate symbol	`codegraph_search(query="json_script")`	`cg query "json_script"`
Read callers	`codegraph_callers(symbol="json_script")`	`cg callers "json_script"`
Read callees	`codegraph_callees(symbol="json_script")`	`cg callees "json_script"`
Analyze change impact	`codegraph_impact(symbol="json_script")`	`cg impact "json_script"` `cg affected django/utils/html.py`
Show file structure	`codegraph_files(path="django/utils")`	`cg files --filter django/utils`
Check index status	`codegraph_status()`	`cg status`

Table 11 maps the two interfaces to the underlying functionalities, using the target function `json_script` from the task `django-15103` as the example. To read its source code, *CC* calls `codegraph_explore(query="json_script")` and *MSA* runs `cgsrc "json_script"` ; both return the source code of the matching symbols with file paths and line ranges, and `codegraph_explore` additionally appends the callers, relationships, and dependencies of every match. To locate a symbol without retrieving the exact context, `codegraph_search` and `cg query` return only its type, location, and signature. For call relationships, `codegraph_callers`/`cg callers` and `codegraph_callees`/`cg callees` return the names and locations of the function's callers and callees. To analyze what code is affected by changing a symbol, `codegraph_impact` and `cg impact` return the symbols that depend on the target symbol, and `cg affected` lists the test files that cover the given source file. To show file structure, `codegraph_files` and `cg files` return the indexed file tree under a directory together with the programming language and the number of symbols in each file. To check index status, `codegraph_status` and `cg status` report the numbers of indexed files, symbols, relations, and whether it is up to date with the working tree.

A.3.2 CodeGraph Usage

Table 12 reports how agents use CodeGraph on Verified-200 and Pro-100. Agents adopt the tool in 74.00–97.50% of tasks. Reading source accounts for 54.16–98.88% of CodeGraph queries and locating symbols for another 0.45–36.48%, which together make up 99.33% and 89.46% of *CC*'s queries on Verified-200 and Pro-100 and 80.84–95.30% of *MSA*'s. Relationship queries (callers, callees, and change impact) receive 0.56–1.60% of *CC*'s queries and 3.24–7.15% of *MSA*'s. File-

Table 12: CodeGraph usage in **RQ2** covering all 100 Pro tasks, including the percentage of tasks in which the agent issued at least one CodeGraph query, and the share of CodeGraph queries by functionality.

	Verified-200				Pro-100			
	CC	MSA_{S46}	MSA_{MM3}	MSA_{Q35+}	CC	MSA_{S46}	MSA_{MM3}	MSA_{Q35+}
CodeGraph task coverage (%)	91.50	95.00	97.50	93.17	77.00	85.00	77.00	74.00
Share of CodeGraph queries (%)								
Read source	98.88	73.97	69.03	58.82	81.47	63.80	64.82	54.16
Locate symbol	0.45	20.65	18.07	36.48	7.99	17.04	19.46	31.51
Read callers	0.56	3.62	4.28	3.30	1.60	3.24	5.36	6.29
Read callees	0.00	0.00	0.00	0.00	0.00	0.00	0.18	0.00
Analyze change impact	0.00	0.49	2.58	0.45	0.00	0.00	1.61	0.47
Show file structure	0.11	1.27	4.35	0.95	8.95	14.79	7.32	7.44
Check index status	0.00	0.00	1.70	0.00	0.00	1.13	1.25	0.14

Table 13: Toolset of the trajectory-analysis agent.

Tool	**Returns**
`get_skeleton`	Step indices, action labels, and truncated actions for a trajectory overview.
`get_step([ids])`	Full thoughts, commands, and command outputs for selected steps.
`get_gt_patch`	The ground-truth patch of the task.
`get_agent_submission`	The agent's final submitted patch.

structure queries are rare on Verified-200 (0.11–4.35%) but more common on Pro-100 (7.32–14.79%), consistent with the larger, multi-language repositories of SWE-bench Pro. Index-status queries are almost never issued.

A.4 Additional Details for RQ3

In this section, we provide additional details on the agent-synthesized and developer-designed skills, including how skills are delivered to the agents, the design of the trajectory analysis agent, the skill synthesis pipeline, and the specific skill contents.

Skill delivery. Agent skills (Anthropic, 2025a) are usually loaded on demand: the agent initially sees only the skill descriptions and reads the full `SKILL.md` once it selects a certain skill, which keeps the context small when many skills are available. Our setting involves no skill-selection decision, because each configuration has a single skill set whose rules apply to every coding task. Following Trace2Skill (Ni et al., 2026), we therefore preload the entire skill set into the system prompt. The preloading is complete rather than partial because our skill sets contain only behavioral rules, with no bundled scripts or resources that would otherwise be discovered on demand. Keeping the skill set always present also yields a controlled comparison, since the rules are available on every task rather than conditionally loaded.

A.4.1 Agent-Synthesized Skills

Trajectory analysis agent design. For the agent-synthesized skills in **RQ3**, we build a lightweight analysis agent that distills corrective rules from each configuration's RQ1 trajectories using its own backbone model. For each trajectory, the agent receives the task outcome, monetary cost, and step count, together with the RQ1 diagnoses, including the flagged cost-inefficient behavior types, their frequencies, and the flagged step indices.

Equipped with four read-only tools (Table 13), the agent verifies flagged actions against the actual step content in the trajectory and finally returns a list of skill candidates, each pairing a confirmed cost-inefficient anti-pattern with a corrective rule. The agent runs as a ReAct loop (Yao et al., 2023) at temperature 0, with a limit of 8,192 tokens per call and 50 steps per trajectory.

Skill synthesis pipeline. Running this agent on every trajectory is costly. In preliminary experiments on MSA_{S46}, the agent tends to read each flagged step and its surrounding context in detail. The number of steps the analysis agent consumes is close to that of the original task-solving trajectory. As a result, the per-trajectory analysis cost is close to the cost of solving the task itself.

To control this cost, we pair the agent with a cheaper single LLM call. We rank each configuration's trajectories by the cost share of their diagnosed cost-inefficient behaviors, analyze the top 25% with

Table 14: Cost of synthesizing skills. **Agent**, **One-time**, and **Merge** are the total costs of the agent analysis, the single-call analysis, and the hierarchical consolidation; **Total**: the total cost of synthesizing skills; **Task cost**: the cost of solving all 300 RQ1 tasks; **Ratio**: Total / Task cost.

Config	Agent ($)	One-time ($)	Merge ($)	Total ($)	Task cost ($)	Ratio
CC	20.33	1.78	0.65	22.76	156.00	14.59%
MSA_{S46}	34.48	2.03	1.08	37.59	166.06	22.64%
MSA_{MM3}	3.72	1.65	0.49	5.86	109.50	5.35%
MSA_{Q35+}	2.92	1.23	0.12	4.27	30.60	13.95%

Table 15: Effects of three approaches on Pass@1, cost, and CoP. SynSkills is reported with and without synthesis cost (Table 14) amortized over Verified-200 and Pro-100 (**+ synthesis**). blue / red indicate robust improvements/regressions, while uncolored cells indicate changes within the run-to-run noise floor; darker shades indicate larger changes.

		Verified-200				Pro-100			
Approach	Metric	CC	MSA_{S46}	MSA_{MM3}	MSA_{Q35+}	CC	MSA_{S46}	MSA_{MM3}	MSA_{Q35+}
CodeGraph	Pass@1 (pp)	$74.17_{-0.83}$	$72.50_{+0.00}$	$75.50_{+0.33}$	$72.50_{+5.33}$	$55.17_{+1.92}$	$51.72_{+1.92}$	$57.47_{-1.15}$	$39.08_{-3.45}$
	Cost ($)	$0.596_{+8.30\%}$	$0.619_{-3.03\%}$	$0.591_{+28.14\%}$	$0.112_{-3.29\%}$	$1.061_{+12.19\%}$	$1.132_{-0.25\%}$	$0.985_{+8.39\%}$	$0.150_{+12.99\%}$
	CoP	$0.804_{+10.22\%}$	$0.854_{-3.04\%}$	$0.782_{+27.52\%}$	$0.155_{-10.63\%}$	$1.923_{+8.69\%}$	$2.188_{-4.32\%}$	$1.715_{+10.20\%}$	$0.388_{+23.20\%}$
SynSkills	Pass@1 (pp)	$74.50_{-0.50}$	$72.00_{-0.50}$	$75.50_{+0.33}$	$69.67_{+2.50}$	$55.00_{+0.67}$	$52.00_{+0.00}$	$62.00_{+2.00}$	$46.00_{+1.00}$
	Cost ($)	$0.541_{-1.38\%}$	$0.496_{-22.32\%}$	$0.462_{+0.15\%}$	$0.115_{-0.70\%}$	$0.826_{-8.86\%}$	$1.006_{-6.13\%}$	$0.766_{-11.90\%}$	$0.132_{-2.22\%}$
	CoP	$0.726_{-0.12\%}$	$0.689_{-21.79\%}$	$0.611_{-0.33\%}$	$0.166_{-4.40\%}$	$1.503_{-9.48\%}$	$1.936_{-6.40\%}$	$1.235_{-15.09\%}$	$0.286_{-4.51\%}$
SynSkills + synthesis	Pass@1 (pp)	$74.50_{-0.50}$	$72.00_{-0.50}$	$75.50_{+0.33}$	$69.67_{+2.50}$	$55.00_{+0.67}$	$52.00_{+0.00}$	$62.00_{+2.00}$	$46.00_{+1.00}$
	Cost ($)	$0.617_{+12.51\%}$	$0.621_{-2.70\%}$	$0.481_{+4.39\%}$	$0.130_{+11.55\%}$	$0.902_{-0.49\%}$	$1.132_{+5.78\%}$	$0.785_{-9.65\%}$	$0.146_{+8.48\%}$
	CoP	$0.828_{+13.88\%}$	$0.863_{-2.04\%}$	$0.637_{+3.88\%}$	$0.186_{+7.38\%}$	$1.641_{-1.18\%}$	$2.176_{+5.52\%}$	$1.266_{-12.93\%}$	$0.317_{+6.01\%}$
DevSkills	Pass@1 (pp)	$73.50_{-1.50}$	$73.00_{+0.50}$	$73.00_{-2.17}$	$69.83_{+2.67}$	$52.00_{-2.33}$	$52.00_{+0.00}$	$62.00_{+2.00}$	$46.00_{+1.00}$
	Cost ($)	$0.473_{-13.94\%}$	$0.372_{-41.73\%}$	$0.387_{-16.03\%}$	$0.103_{-11.34\%}$	$0.894_{-1.51\%}$	$0.733_{-33.00\%}$	$0.801_{-7.88\%}$	$0.140_{+3.57\%}$
	CoP	$0.643_{-11.83\%}$	$0.510_{-42.13\%}$	$0.530_{-13.61\%}$	$0.148_{-14.83\%}$	$1.718_{+3.58\%}$	$1.409_{-33.19\%}$	$1.291_{-11.21\%}$	$0.305_{+1.28\%}$

the agent, and process the remaining affected trajectories with a single LLM call, one trajectory at a time. Beyond the inputs given to the agent, the single-call mode additionally receives a compact representation of the trajectory, in which each step is reduced to its action label together with its thought, action, and output truncated to their first and last lines. Finally, we apply the Trace2Skill (Ni et al., 2026) hierarchical consolidation to merge the per-trajectory rules into one skill set per configuration.

Table 14 reports the synthesis cost of each configuration against the cost of solving the corresponding 300 tasks in RQ1. Table 15 amortizes this cost over the 300 evaluation tasks to the per-task cost of SynSkills, and compares the effects of the three mitigating approaches. Once amortized, the cost advantage of SynSkills largely disappears: the only robust cost reduction that remains is MSA_{MM3} on Pro-100 (-9.65%). MSA_{Q35+} turns into robust increases on both benchmarks (+11.55% and +8.48%), and no setting keeps a robust CoP improvement. Two large savings in Table 3, MSA_{S46} on Verified-200 (-22.32%) and *CC* on Pro-100 (-8.86%), shrink to -2.70% and -0.49% because these two configurations also have the most expensive synthesis cost ($37.59 and $22.76). Because DevSkills incur no synthesis cost, they reduce cost more than the amortized SynSkills in seven of the eight settings, the exception being MSA_{MM3} on Pro-100 (-7.88% versus -9.65%). However, the skill-synthesis surcharge scales inversely with the number of applied tasks. It would fall below 2.00% of the baseline cost if applied over 3,000 tasks. So the amortized picture depends on the deployment scale rather than on the skills themselves.

Agent-Synthesized Skill Content.

Claude Code:

```
## Validation & Testing
- Run each test command **once per code state**. Never re-issue an identical command
    without an intervening code or environment change.
- Before running any test suite, spend one step locating the canonical test runner (check `
    runtests.py`, `pytest.ini`, `tox.ini`, `Makefile`, `README`, or CI config) and the
    correct working directory. Use the right invocation on the first try.
- When a command fails, read the full error, diagnose the root cause once, make a concrete
    corrective change, then retry exactly once. Never re-issue an unchanged failing
    command.
- After a broader suite passes, skip re-running any subset it already covers.
- Capture full test output in a single run (e.g., verbose flags or `tee`), then apply
    multiple grep/analysis steps to that captured result rather than re-executing for each
     filter.
```

```
- When a test failure is confirmed as pre-existing, out-of-scope, or environmental, do not
    re-run overlapping subsets to confirm the same outcome. Move directly to a different
    approach.
- Once a permission barrier is confirmed (e.g., file owned by root, no sudo), stop probing
    it. Immediately pivot to known alternative write paths.

## Reading & Navigation
- Read each source file or region **only once per session**. Before issuing any Read or
    Grep action, check whether the target content was already retrieved and the file is
    unchanged —reuse cached content.
- Consolidate all required context into a single read, or use targeted grep to find needed
    line numbers first, then read once. Only re-read when the file has been externally
    modified or a previously unread section is needed.
- Always read (or confirm you have already read) a file before attempting to edit it. A
    grep/snippet result alone is insufficient —read the relevant section before issuing
    an edit.
- When multiple git commands return no meaningful output, stop using git. Switch
    immediately to reading source files and running reproduction scripts.

## Editing
- Before making any code edit, fully trace all relevant data types, branches, and call
    sites, then write the complete correct replacement in one edit. Avoid incremental
    micro-edits that each leave the code broken.
- Before writing any patch, reason through (a) the exact call site that can receive invalid
     input, (b) the minimal invariant that guards it, and (c) whether fixing the caller vs
    . callee is safer. Commit to one approach and implement it fully.
- Edit source files in-place using the agent's built-in Edit/Write tool, `sed`, or `patch`
    rather than creating intermediate copies or numbered variants.
- When using str_replace or patch, ensure the target string exactly matches existing
    content. After applying any patch, immediately view the modified region to confirm
    correctness before running tests.
- When a library class or symbol is already confirmed by prior codebase analysis, skip a
    live import-check command. Trust prior analysis and proceed directly to editing.

## Script & Artifact Hygiene
- Prefer the project's existing test suite over ad-hoc verification scripts. Only create a
    custom reproduction script if no existing test covers the specific scenario.
- Before writing any script, search the existing codebase first (grep for similar methods,
    check existing tests, inspect the target function's signature). Existing code usually
    reveals the correct approach directly.
- Design a **single comprehensive** diagnostic or reproduction script upfront covering all
    key scenarios (basic case, edge cases, regression). Verify it: (a) imports exist; (b)
    exercises the code under investigation; (c) output distinguishes "bug present" from "
    bug fixed". Overwrite in place rather than creating numbered variants.
- Write multi-line verification scripts to a named temporary file (e.g., `/tmp/repro.py`)
    and execute that file rather than using inline `python -c` commands. Never place it in
     a directory on `sys.path` or name it after a standard library module. Delete it after
     use.
- When a script fails for an environmental reason (missing module, wrong path, missing
    setup), fix the issue in the same script rather than creating a new variant file.
- Use proper assertions (e.g., `assert result == expected`) in verification scripts rather
    than printing and visually inspecting output.
- Do not run scripts to confirm facts already evident from reading source code, or to re-
    confirm behavior already validated by a passing test suite.
- Do not write intermediate analysis or notes to temporary files just to read them back
    immediately. Keep such information in reasoning context.
- When a file-read or directory-listing returns "No such file or directory", verify the
    directory exists first; if it does not, abandon that path entirely rather than
    retrying with different filenames.
```

Mini-SWE-Agent (Sonnet 4.6):

```
## Repository Navigation & Orientation
- Before invoking any test runner, inspect the repository structure once (README, CI config
    , Makefile, tox.ini, pytest.ini, runtests.py) to determine the canonical invocation
    and working directory.
- When a command fails, diagnose the root cause before retrying: distinguish environment/
    config errors from actual test failures.
- When source code inspection makes behaviour clear, skip writing exploratory scripts to
    confirm it; read module source directly with `grep`/`sed` instead of launching an
    interpreter.
- Once a decisive check confirms the required state, stop investigating and report the
    finding; do not re-examine the same evidence from different angles.

## Editing & Patching
- For small, localized edits use targeted in-place tools (`sed -i` with line numbers, `
    patch`, `awk`, minimal heredoc, or precise `str_replace`) rather than read-entire-file
     / write-entire-file cycles.
```

```
- Before any substitution, verify the exact target string (e.g., `cat -A` or `print(repr
    (...))`) to avoid silent no-ops; if two replacement attempts both no-op, switch to a
    fundamentally different method (line-number-based `sed`/`patch`).
- After applying a patch, verify with a single targeted read (`grep -n`, `sed -n 'START,
    ENDp'`, or `git diff`) —not a full file re-read.
- Reserve full-file rewrites for changes scattered throughout the file.
- Before making any speculative change, verify with targeted `grep`/`read` that the change
    is actually needed and that the pattern exists.
- Do not revert a code change unless a concrete test or execution result shows it is wrong;
     never revert speculatively.
- If a file is accidentally emptied or corrupted, immediately restore it from version
    control (`git checkout -- <file>`).
- Avoid splitting files into multiple temporary parts to insert a small change; use a
    single atomic edit to prevent quoting pitfalls and partial-write corruption.

## Script & Artifact Hygiene
- Write one persistent repro/verification script at the start of debugging; consolidate all
     exploratory, diagnostic, and verification logic into it; edit it in place rather than
     creating numbered variants.
- Before creating any new script, check whether the desired change is already present (`
    grep`/`sed`); if so, skip the script entirely.
- For short tasks, prefer `python3 -c` one-liners or heredoc invocations over creating
    named script files.
- When iterating on a script or patch, overwrite the same file —never create numbered
    variants.
- Execute a script immediately after writing it; do not overwrite it multiple times without
     running it.
- When a reproduction script fails to trigger the bug, diagnose the root cause (re-read the
     issue, inspect the code path) before rewriting —write one comprehensive script
    covering all cases.
- Use direct shell commands (`cat`, `sed`, `grep`, `git diff`) to inspect content or
    display diffs rather than writing a temporary Python script.
- When a test fixture or configuration fails, fix the existing fixture in place rather than
     creating a parallel one; reuse existing helpers.
- For non-ASCII content in shell, use heredoc syntax or a temp script file with `
    PYTHONIOENCODING=utf-8` rather than inline `-c` commands.

## Validation & Testing
- Run the relevant test suite exactly once per code-change state; only re-run after an
    intervening code change.
- Capture sufficient diagnostic information in a single well-formed invocation (pipe to `
    tee`, use `grep -E` for summary lines, or use `-v`) so pass/fail status and details
    are visible at once.
- Run the fix-verification script once before the fix (to confirm the bug) and once after (
    to confirm resolution); do not re-run unchanged tests for reassurance.
- Before running a verification script, ensure the fix is complete and handles all known
    edge cases; enumerate edge cases first, then implement and test once.
- When a large test suite consistently times out, stop retrying with minor flag variations;
     run targeted subsets relevant to the change instead.
- When verifying pre-existing failures, a single `git stash →run →git stash pop` cycle is
     sufficient; record the baseline once and compare against it.
- Before running a reproduction script, mentally trace the relevant logic end-to-end and
    fix all visible errors before the first execution rather than iteratively patching and
     re-running.
- Before re-reading a file or region, check whether it has been modified since the last
    read; if unchanged and still in context, skip the re-read entirely.
```

Mini-SWE-Agent (MiniMax-M3):

```
## Validation & test execution
- Plan every test, search, or verification command up front: pick verbosity, output format,
     and filter flags once, then run it once and capture full output (redirect, tee, or a
    generous window).
- Compose invocation with filtering in a single run (`... | grep ...`, `... | tail -n ...`,
     or summary flags) rather than full-suite runs followed by re-runs with different post
    -processing.
- Batch multiple related targets into one invocation; for long runs, redirect full output
    to a file once and apply all subsequent filters to the saved log.
- Trust passing results: skip post-fix sanity probes (import checks, identity checks,
    unrelated module reloads) that don't exercise the changed behavior.
- Don't re-invoke the same suite under cosmetic flag changes, different filters, or "just
    to confirm again." Re-execute only when underlying code or inputs have materially
    changed.
- Scope regression verification to modules that directly exercise the changed path plus a
    small set of clearly related ones; defer broader coverage to the project's own CI.

## Scripts & reproduction artifacts
```

```
- Maintain one canonical reproduction/verification script per debugging concern; evolve it
    in place ——append cases, parametrize via flags/argv, refine assertions ——instead of
    spawning numbered variants.
- Persist reusable reproduction logic in a single small file on first use, then re-execute
    that file rather than fighting inline `python -c` invocations with re-imports and re-
    quoting.
- Consolidate exploratory probes, verification checks, and related cases into one
    comprehensive script or one batched invocation from the start; reserve inline commands
     for genuinely disposable single-statement checks.
- If a heredoc-based script keeps failing on quoting, abandon it for a simpler reproduction
     rather than debugging invisible characters.
- Design repro scripts to be self-contained and leverage existing project infrastructure (
    in-memory setup, inline model definitions, temporary test methods) instead of fragile
    standalone scripts that depend on separately-wired app modules or sys.path hacks.
- Clean up temp files at the end of the task.

## File reading & search caching
- Treat content already read or diffed as cached for the rest of the session: never re-cat,
     re-sed, or re-grep the same unchanged region.
- Rely on `git diff`/`git show` as the authoritative source for post-edit verification;
    prefer one targeted diff or one narrow read over re-catting the patched region.
- Only re-fetch after a confirmed edit, an external change, truncation, or a genuinely new
    line range.
- Once a search enumerates and categorizes its results, record the conclusion and move on;
    if a grep yields nothing relevant, pivot to reading the current code or running the
    failing test instead of repeating with minor keyword variations.

## Editing approach
- For localized edits (a flag, import, block, or method), apply a surgical in-place patch
    ——narrow anchor-based replace, `sed -i`, or a direct edit tool ——rather than
    reconstructing the whole module via heredoc or full-file overwrite.
- Plan the change by checking surrounding context so the edit lands in one pass without
    follow-up rewrites; batch multiple coordinated edits into one atomic patch and dry-run
     against the old content.
- Reserve full-file rewrites for genuinely new files or transformations that genuinely span
     the file.
- On a small follow-up error, surgically fix the differing line in place instead of
    restarting the edit.
- When the fix approach is still uncertain, prototype in a scratch buffer before committing
     to tracked source so speculative edits don't trigger costly round-trips.

## Failure recovery & tooling strategy
- After a small number of failed attempts at the same edit, command, or script, stop
    iterating and diagnose the root cause (wrong path, missing module, environment issue,
    escaping problem) via targeted lookup before retrying.
- Statically analyze the code path or read the full error trace to identify all root causes
     up front and fix them in one revision; avoid the run →see-one-error →fix-one-error
    →run-again cycle.
- If an edit reports no change or fails identically, commit to a different strategy on the
    first failure rather than re-issuing the same command.
- Switch tooling (clean heredoc, environment flags like `PYTHONIOENCODING`) rather than re-
    quoting the same problematic invocation.
- Complete each edit and scratch script in a single direct pass; don't oscillate between
    write →run →revert →write-new loops or iteratively overwrite a scratch file to fix
    typos ——adjust the script's content itself.

## Exploration discipline
- Before broad searches (filesystem-wide grep/find, git history archaeology, API lookups),
    verify the hypothesis with cheap local signals: targeted `git blame`, file-scoped `git
     log`, project-convention check, or a single failing assertion.
- Cap history spelunking at a few focused queries and bound initial exploration to what's
    needed to pick the edit; defer deep architectural investigation (MRO, class
    hierarchies, dunder methods, module maps) until after a first attempt if the simpler
    fix does not suffice.
- Cap broad sweeps (wide `git log --grep`, date-range or keyword variations) to one or two
    attempts before switching strategy; stop searching once the diff has made the fix
    clear.
- Once a fix is implemented, verified against a repro, and passing the relevant tests,
    leave it in place ——don't spend steps grepping for ticket numbers or upstream commits,
     and don't revert a working change to re-derive it from scratch.

## Scope & project conventions
- Define the scope of exploration, edits, and optional artifacts (changelog, docs,
    supplementary files) before starting; avoid applying-then-reverting speculative
    changes.
- Cap setup-only operations at one attempt each (single stash/pop baseline, single sleep
    sized for worst case) and reuse the captured signal for subsequent decisions.
- Before authoring repro or new tests, survey existing project patterns: test layout,
    bootstrap conventions, runner config, and any prior reproducer for the same code path.
- Read existing tests covering the behavior being changed (including invalid-input/negative
     cases) and inspect relevant source to confirm attribute names, API shape, and
```

```
    required imports —so iteration focuses on bug behavior rather than 'AttributeError'/
    missing-registration loops.
- When a third-party package seems required, first check whether it is already installed or
     whether a minimal in-repo stub can exercise the same code path.
- When a task explicitly forbids modifying tests, direct all effort at the source-code fix
    plus minimal verification that the existing suite still passes; don't invest cycles
    computing corrected assertion values or applying-then-reverting test edits.

## Output handling
- When output such as a diff or command result must be both inspected and persisted (e.g.,
    a patch deliverable), do it in a single command using 'tee' or '> file && cat file',
    then reference the saved file at submission.
- Don't write a command's output to a file just to read it back into context —reference
    the in-context display directly; only persist artifacts actually needed for final
    submission.

## API & mocking hygiene
- When a public API exposes a callable-based extension parameter (hook, keep_attrs-style),
    prefer passing the callable over manually reordering positional arguments or post-
    processing the result —the callable form is the documented extension point and avoids
     cascading side effects on inferred output metadata.
- Before writing a reproduction that mocks or stubs a class, read the target method once
    and enumerate every attribute and dunder it references; one targeted read eliminates
    multi-attempt thrashing caused by missing protocol methods discovered after the fact.

## Version control hygiene
- Use selective version-control operations that target only the files intended to change;
    avoid broad patterns that may silently discard or overwrite unrelated prior work on
    other modified files.
```

Mini-SWE-Agent (Qwen-3.5 Plus):

```
## Context & Navigation
- Cache retrieved file content, search results, and verified patterns; reference this
    context instead of re-reading unless the file has been modified or a specific missing
    section is required.
- Navigate directly to target lines or use comprehensive search queries immediately,
    avoiding preliminary reads of unrelated sections or fragmented searches.
- Consolidate verification into a single read operation after applying multiple sequential
    edits to the same region, rather than reading after each individual change.
- Use targeted search commands with specific patterns instead of broadly listing files or
    investigating features not strictly required by the task.

## Editing & Artifact Hygiene
- Modify existing files in-place using targeted, surgical edits (e.g., line-specific
    replacements) instead of rewriting entire files or recreating them from scratch.
- Consolidate all related test scenarios, investigations, and fixes into a single file,
    avoiding the creation of multiple numbered variants, separate scripts, or sequential
    patch artifacts.
- Apply code modifications directly using in-place editing tools instead of creating
    temporary wrapper scripts, intermediate patch files, or numbered variants.
- Prefer direct programmatic editing over chained shell commands for complex changes; if a
    command fails, analyze the error or fall back to manual logic immediately rather than
    retrying broken syntax.
- Plan complete directory layouts and specify explicit text encoding when creating scripts
    to generate all required files in a single pass.
- Verify files are tracked by version control and relevant to the deliverable before
    modifying, skipping build artifacts, cache directories, or untracked files.
- Avoid unnecessary git operations like stash/pop cycles when the working state is correct,
     and review the final diff exactly once after all changes are complete.

## Validation & Testing
- Execute validation commands only once per code state using minimal, focused test cases
    that directly validate specific hypotheses, capturing full output to avoid re-
    execution.
- Consolidate multiple test scenarios, edge cases, verification variants, and diagnostic
    details into a single comprehensive command or script execution.
- When validation fails, diagnose the root cause and extract all necessary information from
     that single run to fix errors immediately, rather than attempting multiple similar
    invocations.
- Establish a baseline for pre-existing failures and verify environment dependencies early
    by running the full test suite once before applying fixes.
- Skip re-running validation, reverting changes, or repeating tests if the underlying code
    has not changed, the state is confirmed via inspection, or static verification
    suffices.
- Once an issue is reproduced or the root cause identified, proceed directly to fix
    implementation without additional reproduction attempts or consecutive verification
    steps.
```

```
- Optimize search and verification by combining related patterns into single operations,
    scoping checks strictly to modified files, and trusting targeted edits succeeded
    unless an error is reported.
- Specify explicit encodings or use automatic handling tools from the first attempt when
    processing source code to prevent encoding-related retries.

## Diagnostic Efficiency
- Prioritize static inspection of source code, diffs, and environment factors to answer
    questions before executing ad-hoc diagnostic commands or creating debug artifacts.
- Consolidate multiple exploratory checks and diagnostic commands into single comprehensive
     scripts or batched invocations, removing temporary artifacts immediately after use.
- Explicitly document key findings from failed iterations to prevent repeating unproductive
     patterns.
- Verify the existence and accessibility of all required output artifacts and ensure the
    test environment is fully initialized before initiating final submission.
```

A.4.2 Full Text of the Developer-Designed Skills

The developer-designed skills are listed below.

```
1. Hypothesis-Before-Read
Before every file read, state in one concise sentence: What you expect to find and why. If
    you cannot state a specific expectation, reason from existing context first. Do not
    read blindly.

2. Reuse Context
Before reading content (`grep / cat / sed / ...`), check whether it is already available in
     the current context. If yes, reuse it instead of fetching it again. Re-read only when
     necessary, and briefly state why (e.g., "confirming exact string before edit", "
    content is too far back in context to rely on"). Avoid issuing multiple overlapping
    reads to retrieve the same context via different tools.

3. Targeted Patch Only
When modifying source code:
- **DO NOT use**: `cat > file << 'EOF'` or `open(f,'w').write("""...""")` to overwrite an
    entire existing file.
- **Use**: Targeted editing, such as `sed -i` or creating your own python script (`content.
    replace(old_substr, new_substr)`, triple-quoted for multi-line).
Write the minimal change. Identify the exact old string and new string before patching.

4. Understand the Test Harness Before Blindly Running Tests
Identify the repository test mechanism before running tests / after running tests with
    wrong commands:
- Check existing documentation / CI configs / test scripts for the correct runner,
    environment variables, and test syntax.
- Treat import, configuration, dependency, and test-loader failures as harness errors, not
    patch failures.
Skip this if you already know the working command.

5. Capture Test Output; Filter, Don't Rerun
When running a test, redirect full output to a file, such as:
```
<test_command> > test_out.txt 2>&1
```
Extract test signal precisely from a prior run instead of re-running the test. Only rerun
    if the source code has changed since the last run.

6. Persist Artifacts; Version Only With Intent
When creating reproduction, test, or patch scripts, save reusable or multi-step logic as a
    workspace file; use inline commands only for simple, one-off probes. Do not create
    numbered variants such as `repro2.py` or `fix_v3.py`; revise the original artifact
    unless separate versions are necessary. State the reason before creating.

7. Break Loops
When repeated reads, tests, or edits produce no new evidence, **Stop.** You may be stuck.
    Do not repeat the same action and state what remains uncertain, form a concrete
    hypothesis, then take one materially different action to move on.

If you are stuck on sub-problems (e.g., verification issues like wrong test environment
    setup), submit your best current solution. A correct patch you cannot fully verify is
    better than a budget exhausted without submission.
```