TOKENECONOMIZER: EXPERIENCE-DRIVEN BLACK-BOX COST CONTROL FOR CODING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Advances in frontier coding agents have sharply increased the token costs of AI-assisted software engineering. Their token efficiency is typically evaluated on isolated tasks, whereas real-world developers use them to continuously build, update, and maintain large-scale projects. This mismatch motivates us to evaluate agents over sequences of realistic coding tasks, where we find that agents exhibit recurring environment-dependent token costs that existing methods struggle to reduce. To minimize these costs, we introduce TOKENECONOMIZER, an external controller that cuts environment-specific token costs over a sequence of tasks. These cuts come from determining (i) which operational micro-skills to inject, (ii) which persistently irrelevant tools to prune, and (iii) when to intervene in unproductive failure loops. Across Claude Code, Codex, and GitHub Copilot CLI paired with frontier models, TOKENECONOMIZER reduces the average token cost by 18.4–30.9% with comparable task success. Savings hold in every repository under every tested agent–model setting and generalize to diverse domains such as sequential web development and security auditing, where costs fall by 6.4–21.2% depending on task structure. Our work establishes adaptive black-box cost control as a promising direction for reducing token costs of coding agents in realistic settings. Our anonymized code is available at `https://anonymous.4open.science/r/TokenEconomizer-anon-1B17`.

1 INTRODUCTION

Frontier large language models (LLMs), coupled with agent harnesses that support tool use (Yang et al., 2024; Wang et al., 2024b; 2025a), have become increasingly capable of completing realistic, long-horizon software engineering tasks (Jimenez et al., 2024; Zhou et al., 2024; Xu et al., 2025; Deng et al., 2025; Merrill et al., 2026; Huang et al., 2026). These components are commonly packaged as commercial products, including Claude Code[1], Codex[2], and GitHub Copilot CLI[3]. These agents are designed primarily for generic task success, but their increased capability comes at a substantial token cost. As shown in Figure 1a, the billed context at each turn includes persistent system and user instructions together with an expanding history of model outputs and tool interactions, causing earlier content to be charged repeatedly across the trajectory.

Token efficiency has primarily been studied and optimized at the level of individual tasks. Model-level controls such as lower reasoning effort save tokens at a cost in task success (Table 1), while harness-level controls such as tool availability and auxiliary instructions typically require manual configuration. More targeted approaches to trajectory compression, observation filtering, and specialized repository exploration often require agent-loop modifications or additional trained components (Xiao et al., 2025; Kovács, 2026; Li et al., 2026b), limiting their plug-and-play deployment. More importantly, task-local optimization overlooks the continuity of software development, where successive changes occur within an evolving codebase (Xu et al., 2025; Joshi et al., 2025; Zhu et al., 2026; Yan et al., 2026). Such continuity creates an opportunity for cost control: within a shared environment, such as a code repository, completed executions can provide reusable experience for future tasks. We therefore

[1] `https://code.claude.com/docs/en/overview`
[2] `https://github.com/openai/codex`
[3] `https://docs.github.com/en/copilot/concepts/agents/copilot-cli`

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.

optimize cumulative cost across a sequence of tasks in that environment, rather than treating each execution in isolation.

We first test whether fixed, externally applied cost-control strategies can deliver consistent token savings. Our preliminary cost profiling covers 60 SWE-Bench tasks across 6 repositories (Jimenez et al., 2024) and evaluates three representative fixed strategies from popular open-source plugins: command-output compression with Rust Token Killer (RTK)[4], semantic code navigation with Serena[5], and structural context injection with Aider's repository map[6]. Empirically, we find that no fixed strategy reduces cost consistently: each has both positive and negative repository-level effects, and their pooled effects are small (Figure 1b). Meanwhile, the default traces reveal three recurring sources of headroom: operational knowledge is repeatedly rediscovered, most loaded tools remain unused, and failed trajectories consume disproportionate cost. Taken together, these findings point to strategies that adapt to each environment from its accumulated traces, rather than fixed strategies applied uniformly.

Motivated by these findings, we introduce TOKENECONOMIZER, an experience-driven external controller for coding agents. It treats each agent as a black box: it acts through exposed prompt and tool settings, monitors execution events, and does not modify model weights or agent-loop code. At its core, a controller applies general cost-control meta-strategies to continually refine environment-specific policies from completed execution traces. These policies adaptively determine (i) which one-line operational facts about the environment to activate as micro-skills, (ii) which loaded but irrelevant tools to prune, and (iii) when to intervene in unproductive failure loops. We evaluate TOKENECONOMIZER with Claude Code, Codex, and GitHub Copilot CLI, paired with frontier models, on 277 SWE-Bench tasks across 6 repositories, 102 Web-Bench tasks across 5 web-development projects, and 200 tasks from two security-auditing benchmarks. Including all controller overhead, our method reduces mean per-task cost on SWE-Bench by 18.4–30.9%, while improving Pass@1 over default execution in 4 of 5 agent–model settings. Cost savings hold in every repository under each tested agent–model setting, and TOKENECONOMIZER variants account for every point on the cost–quality Pareto frontier at or above the default pass rate. On Web-Bench and the security benchmarks, it reduces cost by 16.0–21.2% and 6.4–19.6%, respectively, with Pass@2 and detection F1 remaining comparable to default execution. Our results show that our external cost-control framework can deliver consistent and transferable cost reductions across distinct agent harnesses while maintaining comparable task quality.

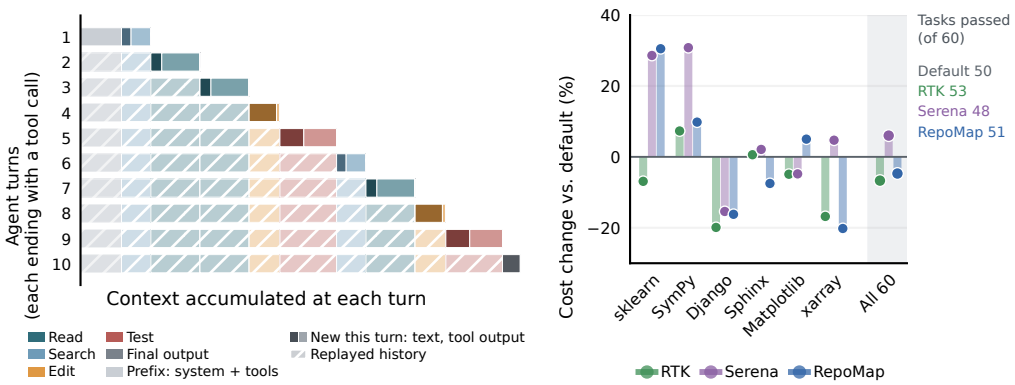
Our contributions can be summarized as follows:

1. We formulate coding-agent cost optimization over sequences of tasks and define a standardized protocol for cumulative end-to-end accounting. Our cost profiling identifies repository-dependent effects of fixed strategies and three actionable sources of headroom.
2. We introduce TOKENECONOMIZER, an external controller that learns environment-specific policies for micro-skill activation, tool pruning, and failure-loop intervention.
3. We evaluate TOKENECONOMIZER across three popular coding agents and four benchmarks spanning 579 tasks, demonstrating cost reductions that hold in every setting we measure, with comparable task success.

2 COST PROFILING AND ANALYSIS OF FIXED COST-CONTROL STRATEGIES

We conduct a retrospective profiling analysis on 60 SWE-Bench tasks from the verified split, with 10 tasks from each of 6 repositories, to test whether fixed strategies save cost consistently and whether completed traces reveal opportunities for adaptive control. As Figure 1a illustrates, every turn bills a persistent system-and-tool prefix together with the accumulated interaction history; caching discounts this replayed context but does not eliminate its cost. Section B provides the detailed protocol.

[4] https://github.com/rtk-ai/rtk
[5] https://github.com/oraios/serena
[6] https://aider.chat/docs/repomap.html



(a) Anatomy of a billed trajectory (schematic)

(b) Cost-saving effects of fixed strategies

Figure 1: Trajectory anatomy and fixed-strategy effects. (a) Schematic of how the persistent prefix and interaction history accumulate across agent turns. (b) Per-repository and pooled cache-aware cost changes for RTK, Serena, and RepoMap relative to default execution (negative is cheaper).

2.1 Repository-specific Effects of Fixed Strategies

We compare the agent's default execution with three fixed strategies, using the same configuration in every repository. *Rust Token Killer (RTK)* rewrites shell commands to compress their outputs; *Serena* adds a semantic code-navigation interface; and Aider's *RepoMap* injects a structural map of repository symbols. Although their outputs may depend on the current task, none learns from completed executions or activates selectively based on prior experience.

Figure 1b shows that every strategy produces mixed repository-level cost changes. For example, all three reduce cost on Django but increase it on SymPy; on scikit-learn only RTK helps, while on Sphinx only RepoMap does. These mixed effects motivate learning repository-specific policies from completed tasks.

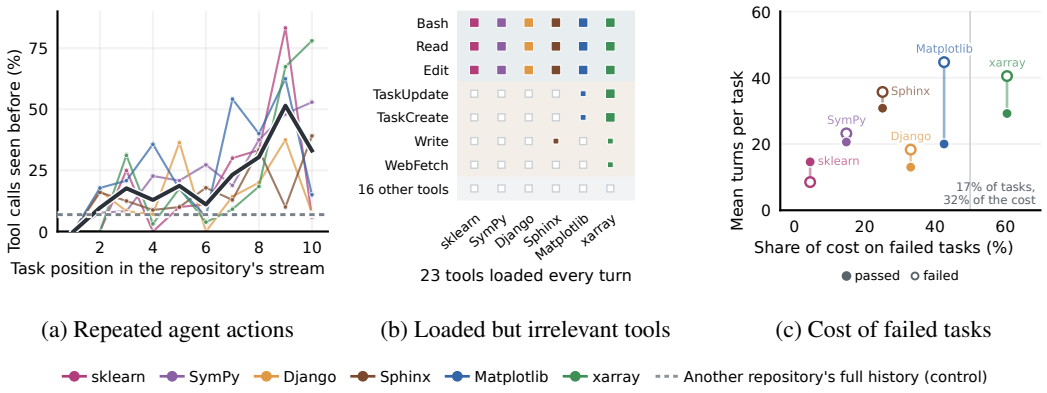
2.2 Three Sources of Removable Cost

Figure 2 identifies three potential sources of cost reduction in traces based on unmodified agent runs.

Operational work repeats within a repository. Actions that recur in at least half of a repository's tasks account for 12–28% of the bytes its tool calls send and receive (Section B describes how calls are matched to actions). The fraction of calls matching an action from an earlier task rises from 14% for the second and third tasks to 30% once at least 5 tasks precede the current one (Figure 2a). Against another repository's complete history, only 7% of calls match, primarily through generic setup and testing commands. This motivates learning repository-specific operational knowledge and injecting it only when relevant.

Most loaded tools remain unused. The agent is shown 23 tool schemas on every turn, yet no repository invokes more than 7 (Figure 2b). Of these tools, 16 are never called and 3 account for 98% of all calls; the remaining tools are sparsely and unevenly used across repositories. A fixed reduced interface could therefore remove shared overhead but also suppress a locally useful tool, motivating evidence-based pruning that can recover a pruned tool.

Failed trajectories consume disproportionate cost. The 10 failed executions constitute 17% of the tasks but 32% of total spend (Figure 2c). A universal turn limit cannot identify them: Matplotlib failures are much longer than successful runs (45 versus 20 turns), whereas Sphinx failures run only slightly longer than its successful runs (36 versus 31). In total, 45 trajectories repeat an earlier failure signature, the normalized first error line of a failing output (Section F), and 51% of the 60 tasks' spend falls after a trajectory's first repetition. Repetition alone does not mark failure, however: 36 of the 50 successful runs also repeat a signature, and more than half of the spend after a repetition falls



(a) Repeated agent actions (b) Loaded but irrelevant tools (c) Cost of failed tasks

Figure 2: Three trace-derived sources of cost-control headroom across the 60 profiled tasks: (a) actions repeated from earlier tasks in the same repository, with another repository's history as a control; (b) loaded tools used by each repository, with frequencies indicated by square sizes; and (c) the cost share and trajectory lengths of failed tasks.

in runs that go on to succeed. A guard must therefore separate persistent loops from recoverable ones, for instance by comparing a run's length with the repository's successful runs, and should redirect before it terminates.

Together, these observations motivate three adaptive decisions, which Section 3 implements as meta-strategies whose policies are learned separately within each environment: which operational knowledge to inject, which persistently irrelevant tools to prune, and when to intervene in an unproductive failure loop.

3 TOKENECONOMIZER

We introduce TOKENECONOMIZER, an external controller that applies three repository-agnostic meta-strategies to learn repository-specific policies from completed traces. Without modifying the underlying agent, it acts through the instruction prefix, loaded tools, and live event stream to synthesize and selectively activate operational micro-skills, prune unnecessary tools, and intervene in unproductive failure loops (Figure 3). Throughout, a *meta-strategy* is one of these three environment-agnostic procedures, a *policy* is the environment-specific decision it learns, and a *fixed strategy*, such as RTK or RepoMap, applies one configuration in every environment.

Our unit of optimization is a chronological task stream $\tau_1, \ldots, \tau_n$ within an environment. By default, an agent runs task τ_i under an instruction prefix P_i, the harness's system instructions followed by the description of τ_i, with every tool in the harness's catalog loaded. When the controller is active, it maintains repository state S_{i-1}, which records the micro-skills and supporting evidence, tool-use history, and turn statistics accumulated from the completed tasks $\tau_1, \ldots, \tau_{i-1}$, with S_0 empty. Before τ_i starts, the controller $\mathcal{C}$ determines the set of active policies based on the task description and current state:

$$(\Delta P_i, \mathcal{U}_i, B_i) = \mathcal{C}(\tau_i, S_{i-1}), \tag{1}$$

The policies given by $\mathcal{C}$ are the set ΔP_i of micro-skills activated at task i, which depends only on S_{i-1} and is rendered as a short manual inserted into P_i between the harness instructions and the task description; the loaded tool set $\mathcal{U}_i$, restricted to catalog tools that remain plausibly useful; and a turn budget B_i that calibrates when an in-flight failure-loop guard may redirect or terminate the rollout. The agent then runs τ_i under these policies, producing a trajectory ξ_i, from which the controller updates the repository state to S_i. The controller may also leave the agent's default configuration in place by activating no micro-skills, keeping the full tool catalog loaded, or never intervening. Its objective is to minimize mean per-task cost, including its own calls, while keeping success near default execution (Section E, Eq. (6)).

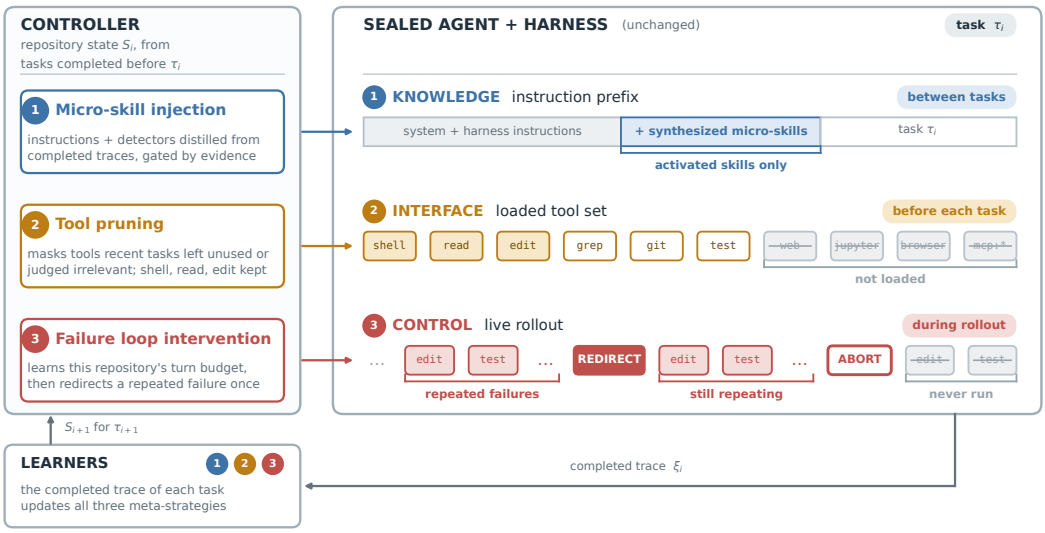


Figure 3: TOKENECONOMIZER applies the same three meta-strategies across repositories, while completed traces refine repository-specific policies over instructions, loaded tools, and live intervention.

3.1 META-STRATEGY I: EXPERIENCE-GROUNDED MICRO-SKILLS

Proposal and evidence collection. The controller periodically condenses a bounded window of completed trajectories into compact digests of operational evidence, such as command templates, test invocations, edited file names, and encountered errors. From these digests, an LLM *proposer* drafts a few candidate micro-skills; its tokens, like those of the controller's other LLM calls, are charged to the task that triggers the call. Each candidate f pairs a simple operational instruction with executable detectors, which are regular expressions or built-in predicates that test the skill against the actual task trajectory. After each task τ_i, the controller runs these detectors on the shell commands and outputs in its trajectory ξ_i. A *scope* detector checks whether ξ_i *exercises* the skill, that is, whether the situation it addresses arose at all; a skill on reproducing issues, for example, is exercised only if the agent attempts to write a reproduction script. An exercised task *contradicts* the skill if it shows the instruction was wrong, such as an import error for a package the skill says is installed, and otherwise *confirms* it. A *need* detector records whether the task needed the skill: the agent either incurred the targeted waste, such as an unnecessary reinstall, or used the stated fact, such as a native test runner; without a need detector, every exercised task counts as needed. These records enter S_i and decide which skills are activated or retired (examples in Section I).

Activation and retirement. A candidate is activated, entering ΔP_i, only while two conditions hold. It must have been confirmed on several tasks whose edited file sets do not overlap, so that its evidence reflects repository-level rather than issue-local recurrence. Its need rate, the fraction of exercising trajectories that needed it, must also stay at or above a threshold. The first condition asks whether a skill is reliable, the second whether repeatedly presenting it is worthwhile. The support of f at task i is

$$q_i(f) = \sum_{j<i} \gamma^{\,i-j} \Big(\mathbf{1}[\xi_j \text{ confirms } f] - \lambda\, \mathbf{1}[\xi_j \text{ contradicts } f] \Big), \tag{2}$$

where $\mathbf{1}[\cdot]$ is the indicator, the decay $\gamma < 1$ discounts older evidence, and the weight $\lambda > 1$ lets one contradiction outweigh one confirmation. A skill that has met the confirmation condition retires once $q_i(f) < 0$, and its identifier is never activated again. The asymmetry is deliberate, because the two errors are not symmetric in cost: a skill that should have been retired is re-billed on every later prefix, whereas a skill retired too eagerly can be proposed again under a new identifier. A skill thus moves from candidate to confirmed to retired (Table 5); a confirmed skill below the need threshold is withheld from ΔP_i but not retired.

3.2 META-STRATEGY II: TOOL PRUNING

Tool pruning reduces the recurring cost of the agent's tool interface by omitting schemas unlikely to help with the current task. The controller always retains core tools for shell execution and file reading, writing, and editing. Other tools are removed after a fixed number of consecutive completed tasks have left them uncalled. In addition, an LLM judge assesses the relevance of each never-called tool not yet removed, using the tool's description and the description of τ_i, in which links to the repository's pull requests, issues, commits, and files are masked. A tool is removed when it meets the unused-history criterion or the judge deems it irrelevant; the remaining tools form $\mathcal{U}_i$ and are loaded for task τ_i. To recover from mistaken exclusions, the controller periodically restores the full tool catalog for a re-audit. A tool called during a re-audit is never masked again, since only never-called tools are masked.

3.3 META-STRATEGY III: FAILURE LOOP INTERVENTION

The controller also derives a repository-specific turn budget from prior trajectories. Let $|\xi_j|$ denote the number of turns in trajectory ξ_j, and let $y_j \in \{0, 1\}$ indicate whether task τ_j succeeded. Let Q_p denote the empirical p-th percentile of the turn counts of successful prior tasks. The budget for τ_i is

$$B_i = \beta \cdot Q_p\big(\{|\xi_j| : j < i,\ y_j = 1\}\big), \tag{3}$$

with p high and a margin $\beta > 1$. Here y_j comes from the benchmark grader, but in realistic settings any comparable success signal can be used. The budget is a reference for identifying unusually long trajectories, not by itself a stopping point. After each tool call, a rule-based check looks for repeated failure signatures, dense error output, or a trajectory that has exceeded B_i, without making a model call. When triggered, an LLM judge reviews a compact digest of recent actions and running counters, including commands, edited file names, and error signatures, to distinguish an unproductive loop from a difficult task that is still progressing. A confident stuck verdict first redirects the agent with a summary of its recent actions and a request to try a different approach. If the failure pattern recurs and the judge again finds the agent stuck, the controller terminates the rollout. Productive or uncertain verdicts let it continue past B_i, up to the forced stop below. As a backstop that the judge cannot veto, a forced stop ends the rollout once it reaches κB_i turns, for a fixed $\kappa > 1$, provided a strict stall rule holds: edit churn, dense errors, and a repeated failure signature at once (Section F).

Section C gives the hyperparameter values, and Section F the implementation details.

4 EXPERIMENTS

We compare TOKENECONOMIZER with default execution, reduced reasoning effort, and fixed strategies on cost and task success across repositories and agent–model settings, then test transfer to web development and security auditing.

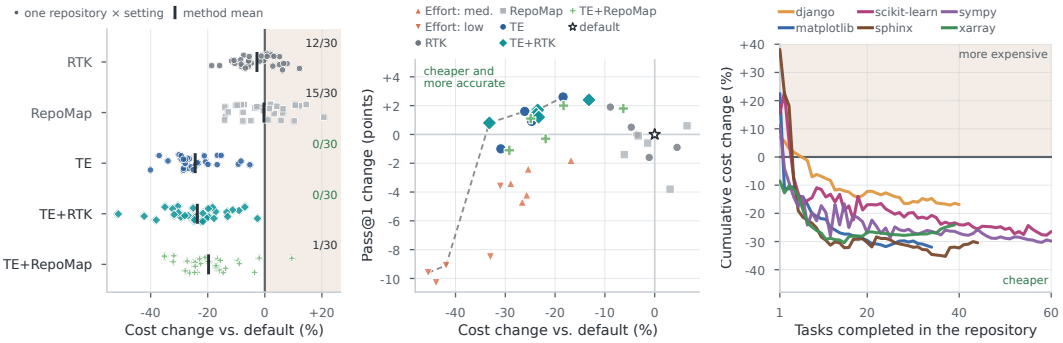
4.1 SETUP

Agents, benchmarks, and methods. We evaluate five agent–model settings: Claude Code with Sonnet 5 and with Opus 4.8, Copilot CLI with Sonnet 5, and Codex with GPT-5.6 Terra and GPT-5.6 Sol. Every setting runs the same 277 SWE-Bench tasks from 6 repositories (Jimenez et al., 2024): scikit-learn (60), SymPy (60), Django (40), Sphinx (44), Matplotlib (34), and xarray (39). These tasks span both the verified split and harder tasks from the full split. Within a repository, tasks run in chronological order, so the controller only learns from work that came earlier in that repository's history. Three additional benchmarks assess generalizability to other domains (Section 4.3): 102 sequential Web-Bench tasks from 5 projects (Xu et al., 2025), and 100 security tasks each from CredData (credential detection) [7] and ReposVul (vulnerability detection) (Wang et al., 2024a). Section D describes how we selected the tasks and streams for each benchmark. In every setting, default execution is the unmodified agent at high reasoning effort, with no external control. We compare it against agent-native execution at medium and low effort, the two fixed strategies RTK and RepoMap, TOKENECONOMIZER, and TOKENECONOMIZER combined with RTK or RepoMap. Serena is intentionally left out due to its conflict with tool pruning and unpromising results observed in our preliminary study.

[7] https://github.com/Samsung/CredData

Table 1: SWE-Bench cost and Pass@1 by method. Cost is the paired relative change from high-effort default execution and includes all controller overhead; Pass@1 is the absolute rate. Every setting runs the full six-repository set with three seeds. All values are percentages; subscripts give the standard deviation across the three seeds. Bold marks, per setting, the highest Pass@1 and the lowest cost among methods whose Pass@1 exceeds default execution; grey costs come with a Pass@1 below default execution.

Method	Claude Code				Codex				Copilot CLI	
	Sonnet 5		Opus 4.8		GPT-5.6 Terra		GPT-5.6 Sol		Sonnet 5	
	Δ Cost ↓	Pass@1 ↑	Δ Cost ↓	Pass@1 ↑	Δ Cost ↓	Pass@1 ↑	Δ Cost ↓	Pass@1 ↑	Δ Cost ↓	Pass@1 ↑
Default	—	$74.7_{\pm 1.3}$	—	$74.0_{\pm 0.6}$	—	$73.8_{\pm 1.0}$	—	$80.1_{\pm 0.8}$	—	$75.0_{\pm 1.5}$
Default (medium effort)	$-26.6_{\pm 1.4}$	$70.0_{\pm 1.8}$	$-16.8_{\pm 2.1}$	$72.2_{\pm 0.4}$	$-25.7_{\pm 2.0}$	$69.6_{\pm 1.2}$	$-28.9_{\pm 1.9}$	$76.7_{\pm 0.7}$	$-25.4_{\pm 2.4}$	$72.6_{\pm 1.2}$
Default (low effort)	$-45.5_{\pm 1.7}$	$65.1_{\pm 0.4}$	$-31.0_{\pm 0.8}$	$70.4_{\pm 0.4}$	$-33.0_{\pm 2.2}$	$65.3_{\pm 1.4}$	$-43.9_{\pm 1.8}$	$69.8_{\pm 0.4}$	$-41.9_{\pm 1.8}$	$65.9_{\pm 0.8}$
RTK	$-1.1_{\pm 2.9}$	$73.1_{\pm 0.6}$	$-3.6_{\pm 3.5}$	$74.0_{\pm 1.4}$	$-4.7_{\pm 2.1}$	$74.3_{\pm 1.1}$	$+4.5_{\pm 2.5}$	$79.2_{\pm 1.2}$	$-8.9_{\pm 3.7}$	$76.9_{\pm 1.0}$
RepoMap	$-1.4_{\pm 1.9}$	$74.1_{\pm 1.5}$	$+6.5_{\pm 2.7}$	$74.6_{\pm 1.5}$	$+3.1_{\pm 2.4}$	$70.0_{\pm 1.8}$	$-6.1_{\pm 2.6}$	$78.7_{\pm 1.6}$	$-3.3_{\pm 2.7}$	$74.9_{\pm 1.3}$
TokenEconomizer	$\mathbf{-23.6}_{\pm 1.8}$	$76.0_{\pm 1.2}$	$-30.9_{\pm 2.1}$	$73.0_{\pm 0.6}$	$\mathbf{-24.7}_{\pm 1.2}$	$74.7_{\pm 1.0}$	$\mathbf{-26.1}_{\pm 2.1}$	$\mathbf{81.7}_{\pm 0.7}$	$\mathbf{-18.4}_{\pm 1.5}$	$\mathbf{77.6}_{\pm 1.3}$
TokenEconomizer +RTK	$-23.5_{\pm 1.2}$	$76.4_{\pm 2.4}$	$\mathbf{-33.2}_{\pm 1.7}$	$74.8_{\pm 1.2}$	$-23.8_{\pm 1.5}$	$\mathbf{75.2}_{\pm 0.9}$	$-23.3_{\pm 1.9}$	$81.3_{\pm 1.4}$	$-13.2_{\pm 2.2}$	$77.4_{\pm 1.1}$
TokenEconomizer +RepoMap	$-18.3_{\pm 3.2}$	$\mathbf{76.7}_{\pm 1.1}$	$-24.9_{\pm 3.9}$	$\mathbf{75.1}_{\pm 0.7}$	$-21.9_{\pm 3.2}$	$73.5_{\pm 1.6}$	$-29.2_{\pm 2.2}$	$79.0_{\pm 1.3}$	$-6.3_{\pm 1.6}$	$76.8_{\pm 0.4}$



(a) Per-environment consistency (b) Cost–quality frontier (c) Cumulative cost change

Figure 4: SWE-Bench results relative to default execution. (a) Cost change in each of the 30 repository–setting cells. (b) Cost and Pass@1 change of each method across the five settings; the dashed line marks the Pareto frontier. (c) Cumulative cost change of TokenEconomizer over each repository's task stream, averaged over the two Claude Code settings.

Evaluation metrics. Our quality metric is Pass@1 under the official grader on SWE-Bench, cumulative Pass@2 on Web-Bench recommended by the original benchmark, and detection F1 on the security benchmarks (more details are included in Section D); efficiency is mean cache-aware cost (Section E, Eq. (4)), which charges every auxiliary controller call to the task that triggered it. Section C lists the controller configuration shared by all benchmarks.

4.2 SWE-Bench Results

Table 1 shows that TokenEconomizer alone reduces mean cache-aware cost by 18.4–30.9% across all five settings. Pass@1 improves in four settings and only falls by 1.0 percentage point on Opus 4.8. Lowered reasoning effort instead reduces Pass@1 by 1.8–10.3 percentage points, and RTK or RepoMap alone saves inconsistently. TokenEconomizer can also be combined with either: both combinations reduce cost in all five settings, and on Claude Code with Opus 4.8, adding RTK raises the saving from 30.9% to 33.2% with Pass@1 above default.

The savings are consistent. Pooled savings can conceal cost increases in individual environments, as the fixed strategies in Section 2 illustrate. Figure 4a reports cost changes for each of the six repositories under each of the five agent–model settings, yielding 30 repository–setting cells. TokenEconomizer reduces cost in all 30, by 5.2–40.1%, as does TokenEconomizer+RTK. TokenEconomizer+RepoMap reduces cost in 29 of 30. By comparison, RTK alone increases cost in 12 cells and RepoMap alone in 15.

Table 2: Cost and quality relative to default execution (%); subscripts are standard deviations over three seeds. (a) Each meta-strategy alone and the full stack on SWE-Bench with Claude Code. (b) TOKENECONOMIZER (TE) on Web-Bench (102 tasks; tasks after an early stop fail; cost paired over tasks both arms reached) and the security benchmarks (100 tasks each); Claude Code averages Sonnet 5 and Opus 4.8.

(a) Meta-strategy ablation

		SWE-Bench	
Model	Method	Δ Cost ↓	Pass@1 ↑
Sonnet 5	Default	—	$74.7_{\pm 1.3}$
	Micro-skills	$-11.8_{\pm 2.1}$	$77.3_{\pm 0.4}$
	Tool pruning	$-13.2_{\pm 1.3}$	$74.3_{\pm 0.5}$
	Failure-loop guard	$-7.1_{\pm 2.2}$	$76.2_{\pm 0.9}$
	Combined	$-23.6_{\pm 1.8}$	$76.0_{\pm 1.2}$
Opus 4.8	Default	—	$74.0_{\pm 0.6}$
	Micro-skills	$-14.3_{\pm 2.3}$	$74.2_{\pm 1.1}$
	Tool pruning	$-17.7_{\pm 1.5}$	$73.5_{\pm 0.6}$
	Failure-loop guard	$-5.4_{\pm 1.8}$	$72.6_{\pm 0.5}$
	Combined	$-30.9_{\pm 2.1}$	$73.0_{\pm 0.6}$

(b) Generalizability to other benchmarks

			Quality ↑	
Workload	Agent	Δ Cost ↓	Default	TE
Web-Bench (pass@2)	Claude Code	$-21.2_{\pm 1.3}$	$69.3_{\pm 1.4}$	$73.3_{\pm 1.6}$
	Codex	$-16.0_{\pm 1.5}$	$64.0_{\pm 1.5}$	$66.5_{\pm 0.7}$
	Copilot CLI	$-18.7_{\pm 1.9}$	$60.8_{\pm 2.2}$	$59.3_{\pm 1.9}$
CredData (F1)	Claude Code	$-19.6_{\pm 0.7}$	$67.9_{\pm 0.5}$	$70.1_{\pm 0.8}$
	Codex	$-16.5_{\pm 1.0}$	$68.0_{\pm 0.4}$	$69.2_{\pm 0.3}$
	Copilot CLI	$-18.3_{\pm 1.4}$	$68.2_{\pm 0.4}$	$67.7_{\pm 0.7}$
ReposVul (F1)	Claude Code	$-7.5_{\pm 1.6}$	$17.5_{\pm 1.1}$	$19.6_{\pm 0.8}$
	Codex	$-6.4_{\pm 1.2}$	$19.4_{\pm 0.9}$	$20.1_{\pm 0.6}$
	Copilot CLI	$-9.3_{\pm 1.6}$	$21.6_{\pm 1.4}$	$22.5_{\pm 0.9}$

Adaptation expands the Pareto frontier. Figure 4b places each method's cost and Pass@1 change relative to default execution in its agent–model setting. Reduced reasoning effort achieves large savings with substantial losses in Pass@1, while RTK and RepoMap stay closer to default and move in mixed directions. TOKENECONOMIZER variants account for every point on the Pareto frontier that maintains or improves the default pass rate, showing that adaptive control reaches a more favorable cost–quality tradeoff across settings.

Savings accumulate over the task stream. TOKENECONOMIZER begins each repository without prior traces, so its benefit may take several tasks to emerge. Figure 4c plots the cumulative cost change of TOKENECONOMIZER (without integrations) relative to default execution within each repository, averaged over the two Claude Code settings. The repositories initially cost more under TOKENECONOMIZER, but all become cheaper by the sixth task and remain cheaper thereafter, ending with cumulative savings of 16.9–32.0%.

Ablations. We run each meta-strategy alone. In the two Claude Code settings, tool pruning carries the largest single share, -13.2% and -17.7%, because it acts on every turn of every task, with micro-skill injection close behind at -11.8% and -14.3% and the failure-loop guard in single digits. Pass@1 stays within 3 points of default for every variant, so none trades task success for cost, and none alone reproduces the full stack's saving (Table 2a; activity counts in Section H).

4.3 GENERALIZATION TO DIVERSE TASK STRUCTURES

To assess generalization beyond SWE-Bench, we evaluate TOKENECONOMIZER on sequential web development (Web-Bench) and two security-auditing benchmarks (CredData and ReposVul). Web-Bench typically involves smaller codebases and workspaces than SWE-Bench, but its tasks remain within one continuous project and naturally inherit earlier workspace changes. In the security benchmarks, each task targets a different repository under a fixed audit prompt, so the procedure recurs without repository continuity. Because no repository recurs, we disable repository-specific micro-skill injection, leaving only two meta-strategies active.

Table 2b shows cost reductions of 16.0–21.2% on Web-Bench and 16.5–19.6% on CredData. The latter result thus demonstrates that the remaining meta-strategies can still achieve substantial savings across new repositories under a shared audit procedure. Reductions are much smaller on ReposVul, at 6.4–9.3%, where shorter trajectories leave less removable cost. The savings thus generalize across these workloads, while their magnitude varies with the reusable structure and cost-reduction headroom within the trajectories.

5 Related Work

Cost reduction for LLMs and coding agents. A classic line of work lowers inference cost by choosing which model answers each query: cascades escalate from cheaper to stronger models until an answer is judged reliable (Chen et al., 2023; Yue et al., 2024; Aggarwal et al., 2024), and routers predict the cheapest adequate model before generation (Ding et al., 2024; Ong et al., 2024). These methods trade answer quality for cost on each query, whereas TokenEconomizer keeps the model and effort fixed and removes cost that recurs within an environment. For coding agents, existing methods reduce cost within a trajectory by controlling context, interfaces, or rollout length. Context-control methods compress prompts (Jiang et al., 2023), shell output with RTK, tool observations (Kovács, 2026), or accumulated trajectories (Xiao et al., 2025); others narrow retrieval through repository maps, knowledge graphs (Vogel et al., 2026), structure-aware scouting (Li et al., 2026b), or trained retrieval agents (Bashir et al., 2026). Interface-level methods restrict or retrieve candidate tools (Paramanayakam et al., 2024; Du et al., 2024), enrich tool interfaces (Liu et al., 2025), or encourage minimal implementations through plugins such as Ponytail[8]. Runtime methods provide corrective feedback (Zhu et al., 2025), early-exit checks (Lu et al., 2025), budget estimates (Lin et al., 2026), or hidden-state failure detection (Ruan et al., 2026). Methods deployable through an agent's exposed interfaces are typically fixed and task-local; our preliminary analysis finds that their effects vary across repositories. TokenEconomizer instead uses completed traces to adapt the instruction prefix, tool set, and stopping policy to each environment.

Cross-task experience for coding agents. Recent benchmarks have started to evaluate agents across continual repair, related-task context reuse, and streaming task sequences (Joshi et al., 2025; Zhu et al., 2026; Yan et al., 2026). Prior methods reuse completed executions as verbal lessons or workflows (Shinn et al., 2023; Zhao et al., 2024; Wang et al., 2024c), or derive repository knowledge from commit histories and repair trajectories (Wang et al., 2026a; Chen et al., 2025). A complementary line of work packages experience into reusable capabilities, including executable skill and tool libraries (Wang et al., 2023; Cai et al., 2024; Yuan et al., 2024), verified programmatic skills (Wang et al., 2025b), refined skill APIs (Zheng et al., 2025), and evolving test-time libraries (Xu et al., 2026). Recent systems further automate skill construction and evolution (Wang et al., 2026b; Ni et al., 2026; Ma et al., 2026; Zhang et al., 2026; Chen et al., 2026; Li et al., 2026a). Because indiscriminate repository context can increase cost without generally improving success (Gloaguen et al., 2026), such experience must be reused selectively. Building on this principle, TokenEconomizer uses cross-task experience to learn cost-control policies applied through agents' exposed interfaces.

6 Limitations and Conclusion

Limitations. Each meta-strategy can still make an incorrect decision: a micro-skill may become stale, a pruned tool may become necessary, and a difficult but productive task may be mistaken for a failure loop. Each decision, however, comes with a recovery path that limits the damage: a stale skill is retired once a trace contradicts it, a wrongly pruned tool returns at the next re-audit, and the failure-loop intervention redirects before it stops and spares uncertain cases. Our experiments cover three widely used coding agents but not more minimal harnesses such as Pi[9]; since TokenEconomizer relies only on the prompt, tool configuration, and event stream, supporting such a harness requires an adapter rather than new meta-strategies. Finally, our SWE-Bench streams consist of related but independent fixes rather than actual user sessions. Continuous sessions should offer more reusable experience (Section A), so interactive coding workflows are a natural next test.

Conclusion. Coding-agent costs accumulate across tasks in recurring environments, which isolated-task evaluations overlook. TokenEconomizer shows that an external controller can learn from completed work to make an unchanged agent cheaper while largely preserving task success, across software engineering, web development, and security auditing.

[8] https://github.com/dietrichgebert/ponytail

[9] https://github.com/earendil-works/pi

AI USE STATEMENT

Generative AI tools were used to assist with implementation, data analysis, and drafting. All generated code and prose were reviewed by the authors. The authors take responsibility for the final content of the paper and its artifacts.

REPRODUCIBILITY STATEMENT

We provide our anonymized code at `https://anonymous.4open.science/r/TokenEconomizer-anon-1B17`, which contains the complete implementation, dataset specifications, and launch scripts that can be used to reproduce our experiments. Relevant details are also mentioned in Section 4.

REFERENCES

Pranjal Aggarwal, Aman Madaan, Ankit Anand, Srividya Pranavi Potharaju, Swaroop Mishra, Pei Zhou, Aditya Gupta, Dheeraj Rajagopal, Karthik Kappaganthu, Yiming Yang, Shyam Upadhyay, Manaal Faruqui, and Mausam. AutoMix: Automatically mixing language models. In *Advances in Neural Information Processing Systems*, 2024.

Hammad Bashir, Kelly Hong, Patrick Jiang, and Zhiyi Shi. Chroma context-1: Training a self-editing search agent. Technical report, Chroma, 2026. URL `https://www.trychroma.com/research/context-1`.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. In *International Conference on Learning Representations*, volume 2024, pp. 54067–54089, 2024.

Lingjiao Chen, Matei Zaharia, and James Zou. FrugalGPT: How to use large language models while reducing cost and improving performance. *arXiv preprint arXiv:2305.05176*, 2023.

Shiqi Chen, Jingze Gai, Ruochen Zhou, Jinghan Zhang, Tongyao Zhu, Junlong Li, Kangrui Wang, Zihan Wang, Zhengyu Chen, Klara Kaleb, Ning Miao, Siyang Gao, Cong Lu, Manling Li, Junxian He, and Yee Whye Teh. Skillcraft: Can LLM agents learn to use tools skillfully? *arXiv preprint arXiv:2603.00718*, 2026.

Silin Chen, Shaoxin Lin, Yuling Shi, Heng Lian, Xiaodong Gu, Longfei Yun, Dong Chen, Lin Cao, Jiyang Liu, Nu Xia, and Qianxiang Wang. SWE-Exp: Experience-driven software issue resolution. *arXiv preprint arXiv:2507.23361*, 2025. URL `https://arxiv.org/abs/2507.23361`.

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

Dujian Ding, Ankur Mallick, Chi Wang, Robert Sim, Subhabrata Mukherjee, Victor Rühle, Laks V. S. Lakshmanan, and Ahmed Hassan Awadallah. Hybrid LLM: Cost-efficient and quality-aware query routing. In *International Conference on Learning Representations*, 2024.

Yu Du, Fangyun Wei, and Hongyang Zhang. AnyTool: Self-reflective, hierarchical agents for large-scale API calls. *arXiv preprint arXiv:2402.04253*, 2024. URL `https://arxiv.org/abs/2402.04253`.

Thibaud Gloaguen, Niels Mündler, Mark Müller, Veselin Raychev, and Martin Vechev. Evaluating AGENTS.md: Are repository-level context files helpful for coding agents? *arXiv preprint arXiv:2602.11988*, 2026.

Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. Deepswe: Measuring frontier coding agents on original, long-horizon engineering tasks. *arXiv preprint arXiv:2607.07946*, 2026.

Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. LLMLingua: Compressing prompts for accelerated inference of large language models. *arXiv preprint arXiv:2310.05736*, 2023.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2310.06770.

Thomas Joshi, Shayan Chowdhury, and Fatih Uysal. Swe-bench-cl: Continual learning for coding agents. *arXiv preprint arXiv:2507.00014*, 2025.

Ádám Kovács. Squeez: Task-conditioned tool-output pruning for coding agents. *arXiv preprint arXiv:2604.04979*, 2026.

Ahmed Lekssays, Hamza Mouhcine, Khang Tran, Ting Yu, and Issa Khalil. LLMxCPG: Context-aware vulnerability detection through code property graph-guided large language models. In *34th USENIX Security Symposium (USENIX Security 25)*, 2025.

Yanzhou Li, Yiran Zhang, Xiaoyu Zhang, Xiaoxia Liu, and Yang Liu. CODESKILL: Learning self-evolving skills for coding agents. *arXiv preprint arXiv:2605.25430*, 2026a. URL https://arxiv.org/abs/2605.25430.

Zhonghang Li, Zongwei Li, Yuxuan Chen, Han Shi, Jiawei Li, Jierun Chen, Haoli Bai, and Chao Huang. Fastcode: Fast and cost-efficient code understanding and reasoning. *arXiv preprint arXiv:2603.01012*, 2026b.

Yuxiang Lin, Zihan Wang, Mengyang Liu, Yuxuan Shan, Longju Bai, Junyao Zhang, Xing Jin, Boshan Chen, Jinyan Su, Xingyao Wang, Jiaxin Pei, and Manling Li. BAGEN: Are LLM agents budget-aware? *arXiv preprint arXiv:2606.00198*, 2026. URL https://arxiv.org/abs/2606.00198.

Kaiming Liu, Xuanyu Lei, Ziyue Wang, Peng Li, and Yang Liu. Agent-environment alignment via automated interface generation. *arXiv preprint arXiv:2505.21055*, 2025.

Qingyu Lu, Liang Ding, Siyi Cao, Xuebo Liu, Kanjian Zhang, Jinxia Zhang, and Dacheng Tao. Runaway is ashamed, but helpful: On the early-exit behavior of large language model-based agents in embodied environments. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 24014–24027, 2025. doi: 10.18653/v1/2025.findings-emnlp.1304. URL https://aclanthology.org/2025.findings-emnlp.1304/.

Yuchen Ma, Yue Huang, Han Bao, Haomin Zhuang, Swadheen Shukla, Michel Galley, Xiangliang Zhang, and Stefan Feuerriegel. Skillgen: Verified inference-time agent skill synthesis. *arXiv preprint arXiv:2605.10999*, 2026.

Mike Merrill, Alexander Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *International Conference on Learning Representations*, volume 2026, pp. 40903–40986, 2026.

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. *arXiv preprint arXiv:2603.25158*, 2026.

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. *arXiv preprint arXiv:2406.18665*, 2024.

Varatheepan Paramanayakam, Andreas Karatzas, Iraklis Anagnostopoulos, and Dimitrios Stamoulis. Less is more: Optimizing function calling for LLM execution on edge devices. *arXiv preprint arXiv:2411.15399*, 2024. URL https://arxiv.org/abs/2411.15399.

Kai Ruan, Zihe Huang, Ziqi Zhou, Qianshan Wei, Jinghao Lin, Xuan Wang, and Hao Sun. Doomed from the start: Early abort of LLM agent episodes via a recall-controlled probe cascade. *arXiv preprint arXiv:2607.06503*, 2026. URL https://arxiv.org/abs/2607.06503.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, 2023.

Martin Vogel, Falk Meyer-Eschenbach, Severin Kohler, Elias Grünewald, and Felix Balzer. Codebase-Memory: Tree-sitter-based knowledge graphs for LLM code exploration via MCP. *arXiv preprint arXiv:2603.27277*, 2026. URL https://arxiv.org/abs/2603.27277.

Boshi Wang, Weijian Xu, Yunsheng Li, Mei Gao, Yujia Xie, Huan Sun, and Dongdong Chen. Improving code localization with repository memory. In *International Conference on Learning Representations*, 2026a. URL https://arxiv.org/abs/2510.01003.

Chenxi Wang, Zhuoyun Yu, Xin Xie, Wuguannan Yao, Runnan Fang, Shuofei Qiao, Kexin Cao, Guozhou Zheng, Xiang Qi, Peng Zhang, et al. Skillx: Automatically constructing skill knowledge bases for agents. *arXiv preprint arXiv:2604.04804*, 2026b.

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

Xinchen Wang, Ruida Hu, Cuiyun Gao, Xin-Cheng Wen, Yujia Chen, and Qing Liao. Reposvul: A repository-level high-quality vulnerability dataset. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, pp. 472–483, 2024a.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents. *arXiv preprint arXiv:2402.01030*, 2024b.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025a.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. *arXiv preprint arXiv:2409.07429*, 2024c. URL https://arxiv.org/abs/2409.07429.

Zora Zhiruo Wang, Apurva Gandhi, Graham Neubig, and Daniel Fried. Inducing programmatic skills for agentic tasks. *arXiv preprint arXiv:2504.06821*, 2025b.

Yuan-An Xiao, Pengfei Gao, Chao Peng, and Yingfei Xiong. Reducing cost of LLM agents with trajectory reduction. *arXiv preprint arXiv:2509.23586*, 2025.

Kai Xu, YiWei Mao, XinYi Guan, and ZiLong Feng. Web-Bench: A LLM code benchmark based on web standards and frameworks. *arXiv preprint arXiv:2505.07473*, 2025.

Weijia Xu, Alessandro Sordoni, Chandan Singh, Zelalem Gero, Michel Galley, Xingdi Yuan, and Jianfeng Gao. Test-time learning with an evolving library. *arXiv preprint arXiv:2605.14477*, 2026.

Dong Yan, Jian Liang, Dapeng Hu, Ran He, Nicholas Jing Yuan, Qi Zhang, and Tieniu Tan. Agentstream: How well do self-evolving llm agents perform under streaming tasks? *arXiv preprint arXiv:2608.00155*, 2026.

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. *Advances in Neural Information Processing Systems*, 2024.

Lifan Yuan, Yangyi Chen, Xingyao Wang, Yi Fung, Hao Peng, and Heng Ji. Craft: Customizing llms by creating and retrieving from specialized toolsets. In *International Conference on Learning Representations*, volume 2024, pp. 40097–40125, 2024.

Murong Yue, Jie Zhao, Min Zhang, Liang Du, and Ziyu Yao. Large language model cascades with mixture of thoughts representations for cost-efficient reasoning. In *International Conference on Learning Representations*, 2024.

Hanrong Zhang, Shicheng Fan, Henry Peng Zou, Yankai Chen, Zhenting Wang, Jiayu Zhou, Chengze Li, Wei-Chieh Huang, Yifei Yao, Kening Zheng, et al. Coevoskills: Self-evolving agent skills via co-evolutionary verification. *arXiv preprint arXiv:2604.01687*, 2026.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024. doi: 10.1609/aaai.v38i17.29936. URL https://ojs.aaai.org/index.php/AAAI/article/view/29936.

Boyuan Zheng, Michael Y Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, Gaowen Liu, Graham Neubig, et al. Skillweaver: Web agents can self-improve by discovering and honing skills. *arXiv preprint arXiv:2504.07079*, 2025.

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

Jiayuan Zhu, Junde Wu, Minhao Hu, Shengda Zhu, Jiazhen Pan, Weixiang Shen, Yijun Yang, Fenglin Liu, Jianye Hao, Yueming Jin, et al. Swe context bench: A benchmark for context learning in coding. *arXiv preprint arXiv:2602.08316*, 2026.

Kunlun Zhu, Zijia Liu, Bingxuan Li, Muxin Tian, Yingxuan Yang, Jiaxun Zhang, Pengrui Han, Qipeng Xie, Fuyang Cui, Weijia Zhang, Xiaoteng Ma, Xiaodong Yu, Gowtham Ramesh, Jialian Wu, Zicheng Liu, Pan Lu, James Zou, and Jiaxuan You. Where LLM agents fail and how they can learn from failures. *arXiv preprint arXiv:2509.25370*, 2025. URL https://arxiv.org/abs/2509.25370.

A Extended Discussion

TokenEconomizer as one instance of a broader class. TokenEconomizer occupies one point in a larger design space: external controllers that apply fixed, environment-agnostic meta-strategies and let completed traces set environment-specific policies. Each of our meta-strategies follows the same pattern, pairing a cheap signal from traces with a conservative action and a way to recover from mistakes: micro-skills are confirmed by detectors and retired on contradiction, pruned tools return through re-audits, and the failure-loop guard redirects before it stops. The same pattern applies to other controls that harnesses expose, such as choosing the reasoning effort or model for each task, compacting context at learned points, or reusing environment setup across tasks, and each strategy could be specialized further to a deployment, for example with longer histories, micro-skills shared across related repositories, or success signals from users rather than benchmark graders. Because the controller needs no model access, retraining, or change to the agent, it can follow agents as they evolve. We therefore see our results as evidence that this direction is promising and underexplored, not as its ceiling.

Generalization without per-environment tuning. A single configuration (Table 4) serves three harnesses, five agent–model settings, and four benchmarks; the only benchmark-specific changes follow from the structure of the streams (Section C). That the savings hold in all 30 SWE-Bench repository–setting cells without tuning suggests that the meta-strategies capture regularities of agentic work, namely recurring operational facts, unused interfaces, and unproductive loops, rather than artifacts of one agent or dataset. The shared configuration is a default rather than a constraint, and its parameters also serve as knobs with a predictable direction: fewer required confirmations or a lower need gate activate micro-skills sooner, a shorter unused streak or a longer re-audit period keeps more tools pruned, and a smaller budget margin β, a lower judge confidence threshold, or a smaller κ lets the guard intervene earlier. Each adjustment trades additional savings against a higher risk of a wrong decision, so users can move along the cost–quality frontier to suit their priorities, much as with reasoning effort, whose savings come with larger losses in Pass@1 (Figure 4b). SWE-Bench is also a conservative test of experience reuse. Consecutive tasks are independent: each starts from a fresh checkout of its own base commit, none inherits the agent's earlier edits, and each stream spans 1.9 to 8.5 years of its repository's history, over which layouts, test commands, and dependencies drift. A

developer's session instead returns to a codebase that changes incrementally, often through the agent's own work, so more of what one task teaches should remain valid for the next. Web-Bench, whose tasks inherit the workspace, is closer to that setting, and TOKENECONOMIZER saves 16.0–21.2% there even though its streams have only 20 or 21 tasks.

Complementarity with fixed strategies. Alone, RTK and RepoMap change Pass@1 inconsistently, by -3.8 to $+1.9$ points across settings (Table 1). Their combinations with TOKENECONOMIZER are more informative. On Claude Code with Opus 4.8, where TOKENECONOMIZER alone is 1.0 point below default, adding RepoMap or RTK raises Pass@1 to 75.1 and 74.8, above default, and a combination attains the highest Pass@1 in three of the five settings. A plausible explanation is that the strategies supply different information. Micro-skills state how to operate in a repository, such as how to run its tests, but by design say nothing about where a fix belongs; RepoMap's ranked signatures address that localization question, and RTK's condensed logs keep failure messages salient in a shorter context. The savings, however, do not simply add up: RepoMap bills its map on every turn, so TOKENECONOMIZER+RepoMap saves less than TOKENECONOMIZER alone in four of five settings, and RTK, whose own savings are inconsistent, adds to TOKENECONOMIZER's saving on Opus 4.8 but lowers it by up to 5.2 points elsewhere. Combining fixed and adaptive strategies thus trades some savings for quality, and which combination is preferable depends on the setting.

Why micro-skills. We call the injected knowledge micro-skills to distinguish it from two forms of reusable experience in prior work (Section 5). Skill and tool libraries package executable code, programs, or APIs for recurring subtasks, while lessons, workflows, and repository memories distill past trajectories into natural-language guidance on how to solve tasks. A micro-skill is narrower than either: a single operational fact about one environment, such as which test runner works, where sources live, or which setup step is unnecessary. Three properties make this granularity suitable for cost control. First, it targets the cost we observe: frontier agents already solve most tasks, and much of their removable spend goes to rediscovering the same environment facts from task to task (Section 2) rather than to missing problem-solving ability. Second, it is continually verifiable: detectors confirm or contradict a one-line claim about commands and outputs on every later trace without a model call, so its validity is rechecked throughout the stream rather than once when it is created, whereas whether a general lesson or procedure helped is hard to read from a trace. Third, it is cheap and low-risk: the manual is billed on every turn, so each entry must be short and earn its place, and because a micro-skill states facts rather than solutions, it carries no task-specific content from earlier tasks into later ones, and a wrong one is retired after little evidence, as Example (c) in Section I shows.

B COST PROFILING PROTOCOL AND DETAILED RESULTS

Cost profiling runs Claude Code with Opus 4.8 at high reasoning effort. From each of the 6 repositories we evaluate 10 tasks, and run default execution along with each of the three fixed strategies once on the same 60 tasks. Each strategy's cost change is relative to the mean cost of default execution. Table 3 reports numerical per-repository results.

Table 3: Cost-profiling results by repository: tasks passed out of 10, and each strategy's cost change relative to default execution on the same tasks.

	Default	RTK		Serena		RepoMap	
Repository	Pass@1 ↑	Δ cost ↓	Pass@1 ↑	Δ cost ↓	Pass@1 ↑	Δ cost ↓	Pass@1 ↑
sklearn	10/10	-6.9%	10/10	$+28.6\%$	10/10	$+30.5\%$	10/10
SymPy	10/10	$+7.3\%$	10/10	$+30.8\%$	10/10	$+9.8\%$	8/10
Django	9/10	-19.9%	9/10	-15.4%	8/10	-16.2%	9/10
Sphinx	6/10	$+0.6\%$	8/10	$+2.1\%$	6/10	-7.5%	8/10
Matplotlib	8/10	-4.9%	9/10	-4.8%	8/10	$+5.0\%$	9/10
xarray	7/10	-16.8%	7/10	$+4.7\%$	6/10	-20.2%	7/10

External cost-saving strategies.

- *RTK* (v0.45.0) is registered as a pre-execution hook on the agent's shell tool. It rewrites each shell command to run through RTK, which condenses the command's output, such as test logs, listings, and diffs, before the agent sees it. The agent's tools and prompt are unchanged.
- *Serena* is an MCP server backed by language servers. It adds symbol-level navigation tools, such as finding a symbol's definition, listing a file's symbols, and finding references. The agent is instructed to prefer these tools over its native file reading and search, which remain available.
- *RepoMap* refers to Aider's repository map. It parses the repository's Python source, ranks files and symbols by their centrality in the cross-reference graph, biased toward identifiers mentioned in the task description, and renders the top-ranked signatures within a 2,048-token budget. The map is generated once per task and supplied as project instructions, so it becomes part of the prefix billed on every turn.

The main experiments use the same RTK version and map budget, and also test TOKENECONOMIZER+RTK and TOKENECONOMIZER+RepoMap to determine whether adaptive savings complement output compression and static repository context. There is no TOKENECONOMIZER+Serena variant, because tool pruning may remove an unused Serena server and so directly confound the two methods; for the same reason, Serena is absent from Table 1.

Headroom measurements. All three measurements read the default-execution traces and estimate exposure to removable cost, which should be interpreted as an upper bound rather than achievable savings.

- *Repeated actions* (Figure 2a). We reduce each tool call to an action key. For shell commands, the key is the command lower-cased, with whitespace collapsed, any leading directory change removed, and only the first 6 tokens kept. For file tools, the key is the tool name and the file's base name. Within each repository, tasks are taken in chronological order, and for each task we measure the share of its tool calls whose key already appeared in an earlier task of the same repository. The cross-task control measures the same share against the complete history of each other repository, averaged over the five.
- *Unused tools* (Figure 2b). We compare the tool catalog the agent receives at session start, which has 23 tools and is the same in every repository, with the tool calls it actually makes, counted per repository.
- *Failed tasks* (Figure 2c). We measure the share of default spend that goes to tasks that fail, and the mean turn count of passed and failed tasks. For the spend after a trajectory first repeats a failure signature, each run's cost is spread evenly over its tool calls, which understates late spend, since every turn re-bills the context accumulated before it.

C CONTROLLER CONFIGURATION

Table 4 lists the controller configuration, and Section J describes how each harness implements it. Within a benchmark, every repository and project uses the same values, and none is tuned per repository, project, or stream. All values were fixed before the main runs, in earlier development runs on the same six SWE-Bench repositories. These values serve six SWE-Bench repositories whose streams range from 34 to 60 tasks and whose default pass rates, turn counts, and tool usage differ widely, as well as five Web-Bench projects that span Web standards and frameworks.

The settings differ across benchmarks only where the structure of the stream requires it. Web-Bench streams have 20 or 21 tasks, too few for the percentile of passed turn counts to warm up before most of the stream has run. The failure-loop guard therefore starts from a pre-registered budget of 30 turns until 5 tasks have passed. On the security benchmarks every task is a different repository, so a repository-specific micro-skill never has a later task in the same repository to apply to; micro-skill injection is disabled. Their tasks are also read-only audits under one fixed brief, so the unused-tool history reflects the audit procedure rather than any single repository, and the judge assesses each candidate against that brief.

Table 4: Controller configuration, shared by all three harnesses.

Config	Value
Micro-skill injection	
Skill sources	proposer
Proposer cadence	every 5 tasks
Confirmation	3 confirmations on non-overlapping edited files
Need gate	rate ≥ 0.4 over ≥ 4 exercises
Support γ, λ	0.8, 5
Tool pruning	
Pruning unit	tool (Codex: capability package)
Removal	unused for 3 tasks, plus per-task judge
Recovery	full catalog every 5 tasks
Failure-loop intervention	
Budget B_i	$1.25\times$ 90th percentile of passed turns
Stuck detection	LLM judge, confidence 0.7
Redirects	1, then stop
Forced stop κ	2.5, if the strict stall rule holds

D Task and Stream Selection

All methods and seeds run the same tasks in the same order, so seeds differ only in the agent's own nondeterminism.

SWE-Bench. We use the six repositories profiled in Section 2, all among the largest in SWE-Bench Verified. Of the 277 tasks, 199 come from SWE-Bench Verified and 78 from the full SWE-Bench test split:

- *Django* (40), *Matplotlib* (34), and *Sphinx* (44) use only Verified tasks. Matplotlib and Sphinx use every Verified task; Django uses 40 of its 231 Verified tasks, all from the start of its Verified history (November 2016 to September 2019), so that consecutive tasks are typically days apart (median 5 days between issue dates) rather than spread over the full 6.7 years.
- *scikit-learn* (60) combines 31 Verified tasks with 29 from the full split.
- *SymPy* (60) combines 41 Verified tasks with 19 from the full split.
- *xarray* (39) has only 22 Verified tasks, and the 13 oldest are spread thinly over 2019 to 2021. Its stream instead uses the 9 Verified tasks from April 2022 onward, interleaved in base-commit order with 30 full-split tasks, one from February 2021 and the rest from March 2022 to January 2023, so that consecutive tasks are typically days apart (median 4 days between issue dates).

We draw the extra scikit-learn and SymPy tasks from the full split to raise the difficulty of the two streams. With Verified tasks alone, nearly every method passes nearly every task in these repositories, so default execution cannot be told apart from its reduced-effort variants. The added tasks have gold patches of 61–200 changed lines and a package version that already appears in the stream; we sort candidates by patch size and sample at even spacing. Because the full split was not validated by human annotators, we checked each added task by applying its gold patch in the official image and running the official evaluation, and every one passed. Each repository's stream runs in base-commit order. Because fix commits are public, every method runs behind an answer-key guard that blocks direct access to the golden patch on GitHub (Section J).

Web-Bench. Web-Bench consists of 50 projects, each a sequence of 20 dependent tasks, and some projects open with an extra `init` task that sets up the scaffold. We run five projects as streams in the benchmark's own task order: calculator (21 tasks), table (21), expressjs (20), react (20), and zustand (20), for 102 tasks in total. Each setting runs every stream with three seeds. We chose the five for diversity across the benchmark's two halves, Web standards and Web frameworks. Calculator and table are built on plain HTML, CSS, and JavaScript, the latter around the HTML `<table>` element. React is a front-end UI framework, zustand a state-management library used with React, and expressjs a Node.js back end with a database. We also excluded projects on which default execution

fails early: the official protocol stops a stream at the first task that fails both attempts, so such a project would leave too short a stream to compare methods on.

We follow the official Web-Bench evaluation protocol (Xu et al., 2025), which allows each task two attempts. If the first attempt fails the project's tests, the agent retries the same task with the test errors as context. The retry matters because each task builds on the previous one: without it, a single failed attempt would end the stream and leave every later task unevaluated. We therefore report Pass@2, the fraction of tasks passed on either attempt, and our cost figures include both attempts. As on SWE-Bench, every method runs behind the answer-key guard (Section J), because the grading specifications are public.

CredData. CredData contains 333 repositories. We exclude the 19 that contain no true secret line: for those, predicting nothing scores an F1 of 1, so they measure only precision. We split the remaining 314 into quartiles by the number of true secret lines per repository, which is heavily skewed (cut points 4, 8.5, and 24.75), allocate the 100 slots in proportion (28, 22, 25, and 25), and select that many repositories from each quartile. The selected repositories contain between 1 and 799 true secret lines (median 8.5).

ReposVul. From the 1,310 instances of the Python split, we drop the 298 whose fix removes no line and so leaves nothing to detect, and the 109 that ReposVul flags as outdated. From the remaining 903 we sample 100 as for CredData, by quartile of vulnerable-line count (cut points 2, 5, and 17; 31, 21, 24, and 24 slots) with a fixed seed. The selected instances have 1 to 411 vulnerable lines (median 5). As on CredData, every task is a different repository under the same fixed audit prompt, and each seed runs all 100. We grade strictly at the line level: a reported location counts only if its file and line match a line the fixing commit removed, and precision is taken over reported lines in the vulnerable files. Prior work instead reports binary F1 over whole functions or code samples on balanced sets (Lekssays et al., 2025); under our metric, naming the right function but a different line scores zero. We keep the strict metric because it mirrors CredData's line-level F1 and needs no further labels.

E Task Stream Formulation and Cost Accounting

Task-stream formalization. Section 3 defines the task stream, the prefix P_i, and the controller map $\mathcal{C}$ of Eq. (1); we reuse that notation here, writing $S_i = F(S_{i-1}, \tau_i, \xi_i, y_i)$ for the state update. The trajectory alternates observations and agent actions, $\xi_i = (o_0, a_0, o_1, \ldots, a_T, o_{T+1})$, so its turn count in Eq. (3) is $|\xi_i| = T + 1$, and its success label is $y_i \in \{0, 1\}$. Because S_{i-1} is formed only from tasks preceding i and the failure-loop guard observes only the prefix $\xi_{i,\le t}$, no decision for τ_i depends on a later task in the stream.

Cost attribution and objective. For task τ_i, let N_i^{u}, N_i^{c}, and N_i^{o} denote the uncached-input, cache-read, and output tokens consumed by the task agent. Let $\mathcal{H}_i$ contain every auxiliary model call attributable to the task, including micro-skill proposal, tool judging, trace judging, and redirect writing. For each auxiliary call h, we record the same three token classes. Controller overhead and total cache-aware cost are then

$$\begin{aligned} H_i &= \sum_{h \in \mathcal{H}_i} \left(w_u N_{i,h}^{\mathrm{u}} + w_c N_{i,h}^{\mathrm{c}} + w_o N_{i,h}^{\mathrm{o}} \right), \\ C_{\mathrm{eff}}(\tau_i; \mathcal{C}) &= w_u N_i^{\mathrm{u}} + w_c N_i^{\mathrm{c}} + w_o N_i^{\mathrm{o}} + H_i, \end{aligned} \tag{4}$$

where the weights are the applicable prices, or their ratios when cost is expressed in fresh-input-token equivalents. We use provider-reported token classes, so cached context is priced as cache-read usage rather than reconstructed as fresh input.

Each auxiliary call is charged exactly once to the task boundary that invokes it. For example, a periodic post-task update is assigned to the task whose completion triggers the update. For the stream

of environment r, the resulting cost and success are

$$\overline{C}_r(\mathcal{C}) = \frac{1}{n}\sum_{i=1}^{n} C_{\text{eff}}(\tau_i;\mathcal{C}),$$
$$\overline{Y}_r(\mathcal{C}) = \frac{1}{n}\sum_{i=1}^{n} y_i. \tag{5}$$

Conceptually, TOKENECONOMIZER seeks to reduce amortized cost while preserving the success of default execution:

$$\min_{\mathcal{C}} \ \mathbb{E}\big[\overline{C}_r(\mathcal{C})\big] \quad \text{subject to} \quad \mathbb{E}\big[\overline{Y}_r(\mathcal{C})\big] \geq \mathbb{E}\big[\overline{Y}_r(\text{default})\big] - \epsilon. \tag{6}$$

The expectation is over stochastic agent execution. Eq. (6) states the design objective rather than an evaluation-time tuning rule: we do not select ϵ after observing the results. Instead, experiments report the observed cost change and success difference relative to matched default executions.

At finalization, task-agent usage and all attributed controller calls are written to the same execution record before Eq. (4) is evaluated. This shared accounting path prevents an optimized method from omitting work moved into an auxiliary call or obtaining an unreported discount through different model routing.

F ADDITIONAL META-STRATEGY DETAILS

Micro-skill injection. The proposer is called after task 4 and every 5 tasks thereafter. Each call sees digests of the repository's 8 most recent tasks, which contain no solution text or gold patches, and returns at most 4 candidates. A candidate is discarded unless it cites at least 2 tasks in the window on which its confirming detector matches. An admitted candidate's detectors are run retroactively over the window's tasks, so it enters with their evidence and can be activated at once. On an exercised task, contradiction is checked before confirmation, and a skill without a need detector counts every exercised task as needed. A skill the proposer marks as high-stakes, such as a test command or a build step, retires at its first contradiction once it has met the confirmation condition. Table 5 summarizes the lifecycle. The three criteria act at different points: the confirmation count moves a candidate to confirmed, the need rate decides on each task whether a confirmed skill is injected, and the support $q_i(f)$ decides retirement. Support is checked only once a skill is confirmed, so a candidate with negative support, which is never injected, stays a candidate; if it is later confirmed, it retires after the next task unless its support has turned positive.

Table 5: Micro-skill lifecycle. Only a confirmed skill can enter ΔP_i, and retirement is permanent.

State	Injected into ΔP_i	Leaves the state when
Candidate	never	confirmed on 3 tasks with non-overlapping edited files $\rightarrow$ confirmed
Confirmed	on each task, if its need rate is ≥ 0.4 over ≥ 4 exercised tasks; otherwise withheld, and rechecked before the next task	$q_i(f) < 0$, or any contradiction of a high-stakes skill $\rightarrow$ retired
Retired	never	never; the proposer may propose a replacement under a new identifier

Tool pruning. A tool's unused streak resets whenever it is called. Before each task, unprotected, never-called tools whose streak has reached three are masked; the other unprotected, never-called tools are sent to the judge, and those it marks irrelevant are masked too; if the judge call fails, they stay loaded. The mask is fixed for the whole task. A re-audit loads the full catalog without calling the judge.

Failure-loop intervention. The rule-based trigger keeps running counters of repeated commands and errors, edit churn, distinct edited files, absence of state change, and repeated verification failures, which the judge's digest also reports. The judge is consulted only while the trigger is open, at most

once every 5 turns; after each productive verdict, this interval grows by 5 turns, up to 20. A forced stop is checked before the judge and cannot be vetoed by it: once the trajectory's turns, counted across a redirect, reach κB_i with $\kappa = 2.5$, and the strict stall rule holds, the rollout ends without a further redirect. A failure signature is the first error line of a failing tool output, such as a traceback, a failed test, or an assertion, with numbers, memory addresses, and file paths removed, so that two outputs that fail the same way match. The strict stall rule holds when three conditions hold at once: edit churn, with at least 4 edits to one file or edits to at least 3 files; dense error output, with at least 6 error outputs; and repetition, with at least 20% of failing outputs repeating an earlier failure signature. The budget in Eq. (3) uses a high percentile of passed turn counts rather than the maximum, so one unusually long success cannot inflate it, while repositories whose successful runs are typically long still receive a proportionally larger budget. The redirect is an LLM-authored summary of the agent's own actions that asks it to try a materially different hypothesis.

G Prompt Templates

This appendix gives the prompts of the controller's model calls, along with the prefix that micro-skill injection adds to the agent's prompt. Slots filled at run time are shown in braces.

G.1 Micro-skill Injection

Proposer. The proposer is called after task 4 and every 5 tasks thereafter, with the digests of the repository's 8 most recent tasks. The prompt's `fired` field is the need detector of Section 3.1.

```
Your task is to mine a durable, repository-specific set of micro-skills for a coding agent
from digests of recently finished tasks in ONE repository. Each digest lists edited files,
command templates, test invocations, and boolean flags, not raw logs.

Emit skills that a FUTURE task in this repository would want stated up front to avoid
wasted commands: the canonical test invocation, whether a reproduction script is redundant,
a required environment variable, a build or reinstall step that is needed (or not needed),
a lint or format hook that reformats, etc. Do NOT emit task-specific fixes or which lines
to change; emit only durable facts about the repository that hold across tasks.

For EACH skill return an object with:
  id           short kebab-case slug
  claim        one sentence stating what is true about this repository
  injection    the exact line to add to the agent's prompt, stated as a declarative fact
               about the repository (where things are, what the canonical command is).
               NEVER phrase it as a prohibition on exploration or tool use: do not tell
               the agent to avoid find/grep/searching/reading or to skip discovering files.
  high_stakes  true if a wrong belief would break the run (test command, build step)
  detector     {scope, confirm, contradict?, fired?}. scope is a regex over commands, or a
               built-in predicate (repro_written, importerror, badarg, ran_tests), that
               selects tasks where the skill is relevant; confirm is a regex whose match in
               commands and output means the skill held on this task; contradict (optional)
               means it was false; fired (optional) means the waste it prevents occurred.
  cite         at least 2 task IDs from the window on which confirm matches

Return at most 4 skills. Be conservative: a skill you cannot ground with a working detector
and two citations is worse than no skill.

If a skill needs evidence that the digests do not contain, first define an extractor that
surfaces it, {name, kind, pattern} with kind in {cmd_regex, out_regex, cmd_count}, and name
it as the detector's scope or fired.

To revise an existing skill, re-emit it with the SAME id and change ONLY
detector.contradict; it keeps its history. Any other change to an existing id is rejected.

Repository: {repository}
Existing skill IDs: {skill IDs}
Existing extractor names: {extractor names}
Task digests (chronological): {digests, as JSON}
Raw command excerpts that no current extractor covers: {commands}

Return a JSON object {"extractors": [...], "skills": [...]}.
```

Micro-skill manual. The manual is rendered before each task from the activated skills and placed ahead of the task prompt; Section I shows individual skills.

```
# repository manual ({repository}), learned from prior tasks in this repository
- {skill 1}
- {skill 2}
...

{task prompt}
```

G.2 Tool Pruning

Tool judge. The tool judge is called before each task other than a re-audit on the unprotected, never-used tools that evidence has not yet masked; the task text is the benchmark's own task description, with links to the repository's pull requests, issues, commits, and files replaced by a placeholder. The agent receives the unmodified description, whose links are unreachable behind the answer-key guard (Section J).

```
You are configuring the toolset for an autonomous coding agent that is about to solve ONE
specific issue by reading, editing, and testing source code in a single repository,
headless and single-shot (no human, one sealed container, no sub-agents, scheduling, or
external memory). Every loaded tool's schema is re-sent on every turn and costs tokens.
Below are the issue and the candidate tools. For THIS issue, mark each tool:
  LOAD  plausibly useful to read, navigate, edit, run, or test code for this issue
  DROP  this issue will not need it
Be conservative: DROP only when you are confident the tool is irrelevant to this issue. Do
NOT assume a tool is needed merely because it exists.

ISSUE: {task text, first 4,000 characters}
TOOLS: {one line per candidate tool: name and description}

Return ONLY a JSON object {tool_name: {"load": true|false, "reason": "<short>"}}.
```

G.3 Failure-Loop Intervention

Both calls read the same digest of the trajectory: its running counters, the last 24 actions, and, once B_i is available, the trajectory's length relative to it. The digest records tool names, commands, search patterns, edited file names, and error signatures, but never file contents or the task text.

Trace judge. The trace judge is called only while the rule-based trigger is open.

```
You are monitoring an AI coding agent that is partway through ONE task in a software
repository. A rule-based trigger flagged its recent process as a possible unproductive
loop. Read the digest of its recent actions and judge whether it is genuinely STUCK
(repeating the same kind of edit, test, or failure with no progress toward passing tests,
and unlikely to recover if it continues the same way) or still making real progress
(changing failures, narrowing the problem, exploring new hypotheses).

Judge ONLY the workflow in the digest; do NOT infer the task's solution or which lines to
change. Favor sparing the agent (stuck=false) when there is any sign of progress; return
stuck=true with high confidence only for a clear, repetitive dead end.

Recent process digest:
turns={t} edit_churn={c} distinct_edit_files={f} error_outputs={e} test_runs={r}
repeated_identical_failures={k} stagnation={s}
Tasks in this repository usually pass within about {B_i} turns; this trajectory is at {t}
turns, {t/B_i}x that length, and its tests are not passing yet. A trajectory far past the
typical length with recurring failures is likely a dead-end loop, even if it touches many
files.
Recent actions (oldest to newest): {last 24 actions}

Return ONLY a JSON object {"stuck": bool, "confidence": 0.0-1.0, "reason": "<short
justification>"}.
```

Redirect writer. The redirect writer is called when the guard intervenes, to write a redirect targeted at the loop.

```
An AI coding agent is stuck in a repetitive, unproductive loop partway through one task in
a software repository. You are given a digest of its recent actions only (tool names, shell
commands, searched patterns, edited file names, and error signatures; never file contents
or the task description). Write a SHORT redirect (2-4 sentences) addressed to the agent
that (1) names the specific repetitive pattern you actually see in the digest (e.g.,
re-editing the same file, re-running the same test to the same failure) and (2) proposes a
materially different next step (a different file or root cause to consider, re-reading the
failing test's expectation, verifying a new hypothesis before editing again).
```

```
Do NOT tell the agent to stop, give up, or reduce effort; only redirect. Do NOT infer the
task's solution, name specific lines to change, or reference any code or identifier you
cannot see in the digest. Output ONLY the redirect as plain text.

Recent process digest: {digest, as for the trace judge}
```

H Ablations of Meta-Strategies

Each single-meta-strategy variant uses the same parameters as the full stack (Table 4), so no variant is tuned for its own ablation. Table 2a reports the cost and Pass@1 of each variant, paired against default execution over the same task–seed units as Table 1. Table 6 reports how often each meta-strategy acted, and Table 7 defines its columns. All columns are per-stream means, but their units differ. *Needed* counts events, one for each exercised task that needed a skill, and is therefore 33 to 37 times the number of confirmed skills beside it. *Masked* is the number of tools still masked at the end of a stream, whereas *Re-audits* counts tasks run with the full catalog, so the two are not comparable. Meta-strategies that a variant disables are reported as zero.

Within each setting, the counts of the combined stack remain close to those of the corresponding single-meta-strategy variant, which indicates that the three meta-strategies seldom interfere with one another's decisions. The counts also show how selectively each meta-strategy acts. About one in five proposed skills is retired within its stream, so the ledger revises its beliefs rather than accumulating them. The trace judge is consulted about 10 to 13 times per stream, but redirects follow on roughly one task in ten and aborts on fewer than one in twenty, since most escalations end in a productive or uncertain verdict.

Table 6: Meta-strategy activity of the ablation variants in the two Claude Code settings, as per-stream means. Table 7 defines the columns, and Table 2a gives the cost and Pass@1 of each variant.

	Micro-skills				Tool schemas		Failure loops		
Method	Proposed	Confirmed	Needed	Retired	Masked	Re-audits	Judged	Redirects	Aborts
Claude Code, Sonnet 5									
Micro-skills	11.4	7.2	268	2.1	0.0	0.0	0.0	0.0	0.0
Tool pruning	0.0	0.0	0	0.0	15.1	3.3	0.0	0.0	0.0
Failure-loop guard	0.0	0.0	0	0.0	0.0	0.0	10.8	4.4	1.4
Combined	12.1	8.4	301	2.6	14.8	3.1	9.9	3.9	2.0
Claude Code, Opus 4.8									
Micro-skills	9.6	6.5	214	1.7	0.0	0.0	0.0	0.0	0.0
Tool pruning	0.0	0.0	0	0.0	14.6	5.2	0.0	0.0	0.0
Failure-loop guard	0.0	0.0	0	0.0	0.0	0.0	13.1	4.8	1.7
Combined	10.3	7.1	241	2.2	14.5	4.3	12.2	5.3	2.2

Table 7: Column definitions for Table 6. Every entry is a mean over the six repository streams of a variant.

Group	Column	Unit	Definition
Micro-skills	Proposed	skills / stream	candidate skills in the stream's final ledger
	Confirmed	skills / stream	candidates that met the confirmation condition and were not retired
	Needed	tasks / stream	exercised tasks that needed a skill, summed over skills
	Retired	skills / stream	retired skills
Tool schemas	Masked	schemas / stream	tools still masked at the last task, not a cumulative count
	Re-audits	tasks / stream	tasks run as a full-catalog re-audit
Failure loops	Judged	consultations / stream	escalations to the trace judge
	Redirects	redirects / stream	redirects written into a trajectory
	Aborts	terminations / stream	trajectories stopped before completion

I EXAMPLES OF META-STRATEGY DECISIONS

This appendix illustrates each meta-strategy with examples from SWE-Bench: micro-skills in (a)–(e), tool masks in (f) and (g), and failure-loop interventions in (h) and (i). Examples (a), (b), and (f)–(i) compare TOKENECONOMIZER with default execution on the same task.

I.1 MICRO-SKILL INJECTION

Each micro-skill example gives the skill as injected, its detectors as the proposer wrote them, and its lifecycle in the stream that learned it. Examples (a) and (b) are activated and injected, and each ends with one task that received the skill, compared with default execution. The remaining three illustrate the other outcomes: (c) is retired by a contradiction, (d) is confirmed but never clears the need gate, and (e) never meets the confirmation condition.

Tasks are numbered by their position in the stream, so the proposer runs after tasks 4, 9, 14, and so on. A detector is a regular expression over the task's commands and outputs unless it names a built-in predicate, such as `ran_tests`, or an extractor the proposer defined, whose pattern is given in parentheses; a dash marks a detector the proposer omitted. In the traces, each line that starts with `$` or a tool name such as `Edit` is one tool call, indented grey text is its output, and a grey count in parentheses stands for tool calls that are not shown.

(a) sympy Claude Code, Opus 4.8 `sympy-13615`

Skill

```
Besides pytest, tests can be run through the native runner `python -c "import sympy;
sympy.test('sympy/.../test_x.py')"` or `python -m sympy.utilities.runtests`.
```

Detectors

```
scope       ran_tests
confirm     sympy\.utilities\.runtests|sympy\.test\(
contradict  -
need        sympy_test_runner (sympy\.utilities\.runtests|sympy\.test\( in a command)
```

Lifecycle, 60 tasks

```
proposed    after task 9, citing tasks 5 and 6
activated   on admission: confirmed on tasks 3, 5, and 6, whose edits do not overlap
            (matexpr.py | permutations.py | sparse.py); need 4/4
injected    tasks 10-60; the traces below are task 11
end         active; 10 confirmations and no contradiction over 10 exercised tasks
```

Default, 15 turns

```
  (10 tool calls: locate and edit Complement, run a reproduction)
$ python -m pytest sympy/sets/tests/test_sets.py -q 2>&1 | tail -15
  /opt/miniconda3/envs/testbed/bin/python: No module named pytest
$ python -m sympy.utilities.runtests sympy/sets/tests/test_sets.py 2>&1 | tail -20
  (deprecation warnings only; no tests run)
$ python -c "import sympy; sympy.test('sympy/sets/tests/test_sets.py')" 2>&1 | tail -20
  tests finished: 72 passed, 2 expected to fail, in 5.48 seconds
  (1 tool call: run the whole sympy/sets package)
```

TOKENECONOMIZER, 11 turns

```
  (9 tool calls: locate and edit Complement, run a reproduction)
$ python -c "import sympy; sympy.test('sympy/sets/tests/test_sets.py')" 2>&1 | tail -15
  tests finished: 72 passed, 2 expected to fail, in 5.37 seconds
```

(b) sphinx Claude Code, Sonnet 5 `sphinx-7889`

Skill

```
The canonical way to reproduce an issue in this repo is to build a minimal Sphinx project
under /tmp with a conf.py and run `python -m sphinx -b <builder> <srcdir> <outdir>`.
```

Detectors

```
scope       python -m sphinx -b
confirm     python -m sphinx -b \S+ \S+ \S+
contradict  -
need        -
```

Lifecycle, 44 tasks

```
proposed    after task 4, citing tasks 1-3
candidate   confirmed on tasks 1 and 3, but both edited a file named conf.py, so the two
            count once toward the 3 confirmations with non-overlapping edits
activated   after task 6, once tasks 5 (inheritance_diagram.py) and 6 (other.py, toctree.py)
            confirm it; need 4/4
injected    tasks 7-44; the traces below are task 15
end         active; 32 confirmations and no contradiction over 32 exercised tasks
```

Default, 11 turns

```
  (7 tool calls: read and edit mock.py, check git log and the installed version)
$ python -c "from sphinx.ext.autodoc.mock import _MockObject; ...; print(obj[T])"
  _MockObject.~T
  (1 tool call: a second inline check)
$ python -m pytest tests/test_ext_autodoc_mock.py -q 2>&1 | tail -20
```

TOKENECONOMIZER, 10 turns

```
  (5 tool calls: read mock.py, search git log, edit, run an inline check)
$ mkdir -p /tmp/genericmock/mymod; cat > /tmp/genericmock/conf.py <<'EOF' ...
    autodoc_mock_imports = ['missingdep'] ... EOF; cd /tmp/genericmock && python -m sphinx
   -b html . _build
  build succeeded, 1 warning.
$ git stash && cd /tmp/genericmock && python -m sphinx -b html . _build ...; git stash pop
  (2 tool calls: run tests/test_ext_autodoc_mock.py, clean up)
```

(c) sympy Claude Code, Sonnet 5 retired after one injection

Skill, high-stakes

```
Run sympy tests with `python -m pytest sympy/<subpackage>/tests/test_<name>.py -q`; this is
the canonical test invocation for this repo.
```

Detectors

```
scope       pytest_module_used (python -m pytest sympy/.*test_.*\.py in a command)
confirm     python -m pytest sympy/.*test_.*\.py
contradict  ImportError|ModuleNotFoundError|no tests ran|error: unrecognized
need        -
```

Lifecycle, 60 tasks

```
proposed    after task 4, citing tasks 2 and 4
confirmed   on admission: tasks 2, 3, and 4, whose edits do not overlap
            (function.py | matexpr.py, summations.py | permutations.py)
activated   after task 5, once the need gate has 4 exercised tasks
injected    task 6 only
retired     after task 6, whose trace contradicts it, which drives its support below zero
end         retired; 40 confirmations and 11 contradictions over 51 exercised tasks
```

TOKENECONOMIZER, task 6, `sympy-13031`, 15 turns

```
  (11 tool calls: locate the stacking code, run reproductions, read git history)
$ python -m pytest sympy/matrices/tests/test_commonmatrix.py -q -k "hstack or vstack" 2>&1
    | tail -20
  /opt/miniconda3/envs/testbed/bin/python: No module named pytest
$ source activate testbed 2>/dev/null; which python; python -c "import pytest" 2>&1; conda
    env list
  ModuleNotFoundError: No module named 'pytest'
  (1 tool call: run the tests with bin/test)
```

The instruction is wrong for this environment, which has no pytest: all 57 traces in the stream that invoke pytest fail with `No module named pytest`. The skill was nonetheless activated, because its confirming detector matches the pytest command itself rather than a passing result, so the agent's failed attempts on tasks 2–5 counted as confirmations. The first task to receive the skill tried the command and then probed for the module, and the probe's `ModuleNotFoundError` matched the contradicting detector. Weighted by $\lambda = 5$, that single contradiction outweighs the decayed confirmations, and the skill left the manual after one injection. Example (a) shows the native runner that another stream learned for the same repository.

(d) sympy Claude Code, Opus 4.8 never clears the need gate

Skill, high-stakes

```
The sympy package is imported from the in-tree source, so edits to .py files take effect
immediately; no `pip install`/`setup.py develop` reinstall is required before testing.
```

Detectors

```
scope       ran_tests
confirm     python -m pytest\s+sympy/|sympy\.test\(
contradict  pip install|setup\.py\s+(develop|install)
need        reinstall_invoked (pip install|setup\.py\s+(develop|install) in a command)
```

Lifecycle, 60 tasks

```
proposed    after task 9, citing tasks 3 and 6
confirmed   on admission: tasks 3, 5, and 6, whose edits do not overlap
            (matexpr.py | permutations.py | sparse.py); need 0/4
activated   never: no exercised task ever reinstalls, so the need rate stays 0 < 0.4
end         not activated; 10 confirmations, no contradiction, need 0/10
```

This skill comes from the same stream as (a) and has the same confirmations. Its confirmations hold, but the agent never attempts the reinstall it warns against, so injecting it would add its tokens to every turn without preventing any waste.

(e) sphinx Claude Code, Sonnet 5 never activated

Skill, high-stakes

```
The canonical test file for sphinx/domains/python.py is tests/test_domain_py.py;
Python-domain edits are verified with `python -m pytest tests/test_domain_py.py`.
```

Detectors

```
scope       domains/python\.py|test_domain_py
confirm     test_domain_py\.py
contradict  test_domain_py\.py.{0,20}(FAILED|ERROR|not found)
need        -
```

Lifecycle, 44 tasks

```
proposed    after task 34, citing tasks 30 and 34
candidate   confirmed on tasks 30, 33, 34, 38, 39, 40, and 43, which edited
            python.py | docfields.py | python.py | __init__.py, inspect.py, python.py |
            python.py | python.py | python.py
end         candidate; 7 confirmations, only 2 with non-overlapping edits
```

Every confirmation except one comes from a task that edited a file named `python.py`, so the evidence reflects one module that recurs across tasks rather than a practice that holds across the repository, and the skill is never activated.

I.2 Tool Pruning

Examples (f) and (g) show the tool mask over the first 4 tasks of the Django stream on Claude Code and Copilot CLI, together with the prompt tokens of each task's first request.

(f) django Claude Code, Sonnet 5 first 4 tasks tool pruning

Masked tools

```
AskUserQuestion, CronCreate, CronDelete, CronList, EnterPlanMode, EnterWorktree,
ExitPlanMode, ExitWorktree, NotebookEdit, ScheduleWakeup, Skill, Task, TaskCreate, TaskGet,
TaskList, TaskOutput, TaskStop, TaskUpdate, Workflow
```

Default, every task

```
  23 tools loaded; first request of each task: 42-46k prompt tokens
```

TokenEconomizer, first-request prompt tokens (default in parentheses)

```
task 1  django-10097  23 tools loaded             42,866 (42,866)
task 2  django-10554  judge masks the 19 above    16,295 (43,351)
task 3  django-10880  judge masks the same 19     15,003 (42,059)
task 4  django-10914  evidence masks the same 19  15,299 (42,355)
                      (each unused for 3 tasks)
  loaded from task 2 on: Bash, Edit, Read, Write
```

(g) django Copilot CLI, Sonnet 5 first 4 tasks tool pruning

Masked tools

```
factories_manage, fetch_copilot_cli_documentation, list_agents, read_agent, run_factory,
session_store_sql, skill, sql, task, write_agent
```

Default, every task

```
  19 tools loaded; first request of each task: 25-26k prompt tokens
```

TOKENECONOMIZER, first-request prompt tokens (default in parentheses)

```
task 1  django-10097  19 tools loaded             25,765 (25,760)
task 2  django-10554  judge masks the 10 above    11,347 (26,249)
task 3  django-10880  judge masks the same 10     10,066 (24,969)
task 4  django-10914  evidence masks the same 10  10,343 (25,248)
                      (each unused for 3 tasks)
  loaded from task 2: bash, create, edit, glob, grep, list_bash, read_bash, stop_bash, view
```

I.3 FAILURE-LOOP INTERVENTION

Example (h) shows a redirect after which the agent tests its change and finishes in fewer turns than default execution, and (i) shows a forced stop on a task that default execution also fails.

(h) sphinx Claude Code, Opus 4.8 `sphinx-11445` redirect

Default, 42 turns

```
  18 edits to sphinx/util/rst.py and 3 test runs before the task passes
```

TOKENECONOMIZER, 31 turns

```
  turns 1-18: 11 edits to sphinx/util/rst.py, no test run, the same failure repeated 3 times
Redirect (trace judge: stuck, confidence 0.72)
  Process check (automated): so far you've made 11 edits across 1 file(s) ... and run 0
  verification(s) ... try a MATERIALLY DIFFERENT hypothesis: re-read the failing test's
   expectation ...
$ grep -rn "prepend_prolog\|rst_prolog\|docinfo" /testbed/tests/test_util_rst.py
$ python -c "from sphinx.util.rst import prepend_prolog; ..."
$ python -m pytest tests/test_util_rst.py -q 2>&1 | tail -15
  7 passed in 0.65s
```

(i) xarray Claude Code, Opus 4.8 `xarray-6798` forced stop

Default, 32 turns, fails

```
  6 edits to xarray/core/coordinates.py and 3 test runs; the task still fails
```

TOKENECONOMIZER, 25 turns, fails

```
  learned turn budget B_i = 10.25 turns; forced stop at 2.5 B_i
  by then: 7 edits to xarray/core/coordinates.py, 14 error outputs, the same failure
   repeated 3 times
Stop: past 2.5 B_i, and the strict stall rule fires
  the task fails, as it does under default execution, 7 turns sooner
```

J HARNESS-SPECIFIC CUSTOMIZATION

TOKENECONOMIZER reaches each agent only through interfaces its harness already exposes: the task prompt, the tool configuration applied at launch, and the event stream the harness emits while it runs. The three meta-strategies and normalized trace format are shared across harnesses, and harness-specific adapters implement the available controls. Every paid controller call uses the same model and routing as the task agent.

Vendor-specific event streams are converted into a common representation containing tool names, arguments, observations, model outputs, usage, and grading outcomes. Meta-strategy implementations consume only this normalized interface. All experiments use Claude Code 2.1.170, Codex 0.153.1, and GitHub Copilot CLI 1.0.84-7.

Answer-key guard. Web-Bench grading specifications and SWE-Bench fix commits are public on GitHub, so an agent with network access could read the answer instead of solving the task. On both benchmarks, every method, including default execution, therefore runs behind the same guard on all three harnesses: the task container resolves GitHub hosts to an unreachable address, so any command that looks them up by name fails to connect. Claude Code and Copilot CLI additionally run a pre-execution hook that refuses shell commands referencing GitHub; Codex 0.153.1 accepts a

hook configuration but did not invoke it in our tests, likely a limitation of this version, so it relies on the first layer alone.

Claude Code. Claude Code runs headless and emits a JSON event stream, which the controller reads live and keeps as the task's trace. The tool catalog comes from the stream's session-start event.

- *Micro-skills.* The rendered manual is prepended to the task prompt.
- *Tool pruning.* Masked tools are passed to the CLI's disallowed-tools option, which removes their schemas from every request.
- *Failure-loop intervention.* The guard reads the stream as it arrives. To redirect, it stops the CLI and starts a new invocation in the same container, with the redirect in its prompt and the agent's edits still on disk. A second stuck verdict ends the task. Token usage from both invocations is summed.

Codex. Codex also runs headless with a JSON event stream. It differs from the other two harnesses in one interface: its tools cannot be removed one at a time, because groups of related tools and instructions can only be switched on or off in its configuration file.

- *Micro-skills.* The rendered manual is prepended to the task prompt, as on Claude Code.
- *Tool pruning.* The pruning unit is a capability package rather than an individual tool: sub-agent collaboration, goal tracking, image input, bundled skills, and the base prompt's guidance for interactive chat, which a headless run never uses. The rules of Section 3.2 apply to packages unchanged. The mask is written into each task's configuration file.
- *Failure-loop intervention.* As on Claude Code, a redirect starts a new invocation in the same container, and a second stuck verdict ends the task.

Copilot CLI. Copilot CLI's event stream does not report token usage, so the controller reads usage and the tool catalog from the CLI's debug log.

- *Micro-skills.* The rendered manual is prepended to the task prompt, as on Claude Code.
- *Tool pruning.* Masked tools are passed to the CLI's excluded-tools option, which removes their schemas from the request; its deny-tool option would only refuse execution while still billing each schema on every turn.
- *Failure-loop intervention.* As on Claude Code, a redirect starts a new invocation in the same container, and a second stuck verdict ends the task.