VERIFICATION AS A POLICY: META VERIFICATION FOR AI AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Verifying whether an AI agent has successfully completed a task could require inspecting its execution trajectory, running executable checks, or consulting other models. Which verification action to perform next depends on the evidence collected so far and the resources remaining. We study *Meta-Verification*, formulating verification as a sequential decision-making problem in which an agent adaptively selects verification actions and decides when to stop under a resource budget. Rather than following a prescribed verification procedure, we propose Meta-Verification, an approach which provides an agent the ability to construct an instance-specific verification procedure and delegate compute for executing it as evidence accumulates. We introduce *AgentVerificationBench*, comprising 541 best-of-16 coding tasks and 189 best-of-3 scientific tasks, to evaluate verification accuracy, realized cost, and budget adherence. We find that frontier agents combine diverse verification mechanisms, but the benefits of online replanning vary across models, and unmodified agents frequently exceed their nominal budgets. We therefore investigate harness optimization as a means of improving Meta-Verification through experience while keeping model weights fixed. On a held-out science test set, harness optimization reduces mean verification cost from \$23.20 to \$5.72 and increases budget adherence from 42.3% to 98.7%, while selection accuracy changes from 60.3% to 55.1%. These findings establish sequential verification-policy construction as a measurable agent capability and demonstrate that its resource efficiency can be substantially improved through harness optimization.

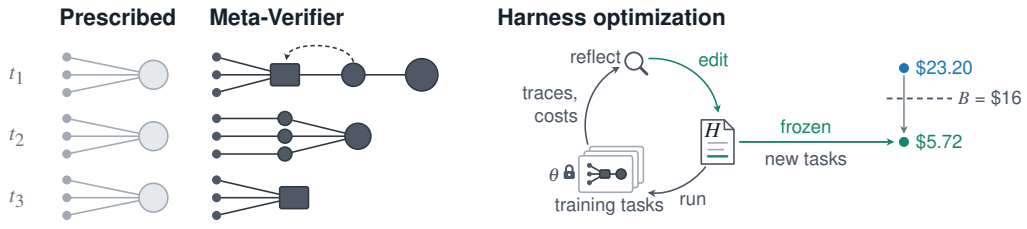

Figure 1: **Meta-Verifiers adapt verification procedures to each task under a budget.** Optimizing the reusable harness using feedback from solving previous verification tasks, while keeping model weights fixed, improves accuracy-cost trade-offs and budget control on verification benchmarks.

1 INTRODUCTION

As AI agents increasingly undertake complex, multi-step tasks (Anthropic, 2025; OpenAI, 2025; Google, 2026), determining whether they have successfully completed their objectives is becoming a fundamental challenge. Verifying an agent could require inspecting its execution trajectory, examining task artifacts, running executable checks, or consulting other models (Zhuge et al., 2024). For example, verifying a coding agent's solution could involve examining its code changes, checking the resulting repository state, and running targeted tests. The appropriate checks depend on the task and the evidence already available: a simple test could establish correctness in one case, while an unexpected failure could require further investigation in another. This challenge becomes more pronounced when

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.

test-time compute is used to generate multiple candidate trajectories and a verifier must identify a successful one (Brown et al., 2024; Snell et al., 2025; Setlur et al., 2025).

Existing verification methods offer a range of mechanisms, including direct language-model judgments (Zheng et al., 2023), model-based verification (Kwok et al., 2026a), pairwise selection (Singh et al., 2026), and tool-based evaluation (Zhuge et al., 2024; Han et al., 2025). Some methods adapt their verification effort based on intermediate results, but typically within a prescribed mechanism or workflow. Verifying complex agent behavior could instead require choosing among different mechanisms as evidence accumulates. A failed test could motivate trajectory inspection, conflicting model judgments could warrant an executable check, and decisive evidence could make further verification unnecessary. The central challenge is therefore not only how to assess correctness, but also which verification action to perform next and whether its expected value justifies its cost.

We study *Meta-Verification*, framing verification as sequential decision-making under a resource budget. Given a task, one or more candidate trajectories, and available verification resources, a *Meta-Verifier* selects its next action based on accumulated evidence and remaining resources. Its available actions include inspecting trajectories, executing checks, delegating to auxiliary verifier agents, comparing candidates, and terminating with a final decision. The instance-specific verification procedure emerges from this sequence of decisions rather than being completely specified before execution. To investigate the value of online replanning, we compare *Adaptive Meta-Verification*, which permits revising the procedure as evidence arrives, with *Plan-then-Execute*, which commits to a procedure in advance. The latter permits conditional branches, but not subsequent replanning.

To evaluate this capability, we introduce *AgentVerificationBench* (AVB), comprising 541 best-of-16 coding verification tasks from Terminal-Bench (TB) 2.1 (Merrill et al., 2026) and DeepSWE (Huang et al., 2026), and 189 best-of-3 scientific verification tasks from Terminal-Bench-Science. The coding tasks require selecting a successful trajectory from a mixture of successful and unsuccessful candidates; the scientific tasks additionally include cases in which every candidate is incorrect and must be rejected. Using fixed candidate sets, executable correctness labels, and controlled verification resources, AVB evaluates selection accuracy, measured verification cost, and budget adherence.

Our experiments show that frontier agents construct diverse, instance-specific verification procedures by combining executable checks, trajectory inspection, delegation, and candidate comparison. However, the benefits of online replanning vary across models and evaluation settings. More consistently, unmodified Meta-Verifiers exhibit poor resource control, frequently exceeding their nominal budgets despite explicit budget instructions. Thus, selecting useful verification actions does not necessarily imply controlling their cumulative cost.

We therefore investigate whether Meta-Verification can be improved through experience without updating model weights. Using feedback from previous verification tasks, we optimize the reusable agent harness, including its prompts, tools, skills, memory, and orchestration (Agrawal et al., 2026; Zhang et al., 2026b; Lee et al., 2026). The optimized harness is subsequently applied to unseen tasks without access to their ground-truth outcomes. On the held-out 78-task AVB-Science test set, harness optimization reduces mean verification cost from \$23.20 to \$5.72 and increases budget adherence from 42.3% to 98.7%, while selection accuracy changes from 60.3% to 55.1%. These findings establish substantial gains in resource efficiency, although they do not isolate improved online replanning from more economical reusable verification routines.

Our main contributions are:

- We formulate Meta-Verification as sequential verification-policy construction under resource constraints and investigate the value of online replanning (Sections 3.1 and 4.2).
- We show that online replanning gains depend on the model by comparing Plan-then-Execute and Adaptive Meta-Verification (Sections 3.1 and 4.2).
- We introduce AgentVerificationBench to evaluate the verification of coding and scientific agent trajectories using selection accuracy, realized cost, and budget adherence (Section 3.2).
- We demonstrate that reusable harness optimization with fixed model weights can reduce mean verification cost at least $4\times$ using experience from previous tasks (Sections 3.3 and 4.3).

2 Related Work

Verification and test-time scaling. Verification plays an important role in assessing generated solutions and realizing the benefits of test-time compute scaling (Brown et al., 2024; Snell et al., 2025; Setlur et al., 2025). Prior work has explored self-consistency and search over reasoning or action sequences (Wang et al., 2022; Yao et al., 2023; Zhou et al., 2023), as well as learned verifiers for identifying correct solutions (Cobbe et al., 2021). Process- and outcome-based supervision (Lightman et al., 2024; Uesato et al., 2022) and generative verification (Zhang et al., 2025) provide additional correctness signals. Self-verification (Weng et al., 2023), iterative self-refinement (Madaan et al., 2023), and tool-interactive critiques (Gou et al., 2024) use feedback to assess or improve generated solutions. While these approaches establish the importance of verification for generation and reasoning, verifying complex agent trajectories introduces an additional challenge: determining which evidence to gather and which verification actions to perform.

Adaptive and agentic verification. Existing approaches provide a range of mechanisms for assessing candidate solutions, including direct LLM judgments (Zheng et al., 2023; Kim et al., 2024; Zhu et al., 2025), model-based verification (Kwok et al., 2026a), and pairwise selection (Singh et al., 2026). Other approaches use structured tool-based verification (Zhuge et al., 2024; Han et al., 2025), verification-guided model routing (Aggarwal et al., 2024), and ensembles of weaker verifiers (Saad-Falcon et al., 2025). Related work also investigates verification scaling for vision-language-action alignment (Kwok et al., 2026b). Several of these methods adapt their verification effort or choice of actions using intermediate results. However, their adaptation is generally organized around particular evaluation mechanisms or prescribed verification workflows. Meta-Verification instead studies an agent's ability to select and combine available verification mechanisms as evidence accumulates. The Meta-Verifier's available actions include inspecting trajectories, executing checks, delegating to different models, comparing candidates, and revising its verification approach during execution. We evaluate this capability under controlled resource access, measuring decision accuracy together with realized verification cost and budget adherence.

Harness optimization and learning from experience. Recent work improves agent behavior by optimizing the surrounding harness while keeping model weights fixed (Hu et al., 2025; Zhang et al., 2026a). Agrawal et al. (2026) optimizes prompts through feedback, while structured playbooks (Zhang et al., 2026b) provide another mechanism for incorporating experience into agent behavior. Meta-Harness (Lee et al., 2026) allows optimization across harness components, and subsequent approaches use execution feedback to guide harness revisions (Lin et al., 2026; Park et al., 2026). More directly related to verification, Rosset et al. (2026) automate the design of verifiers for computer-use agents. We investigate harness optimization in the context of sequential verification-policy construction. Rather than optimizing a prespecified verification primitive, our optimizer can revise prompts, tool-use code, skills, memory, and orchestration to change how the Meta-Verifier selects checks, allocates resources, and decides when to stop. The optimized harness is reused across tasks, while each task induces its own sequence of verification actions based on the evidence encountered during execution. Our experiments evaluate the resulting correctness–cost trade-offs and changes in verification strategies, without attributing improvements specifically to stronger online replanning rather than more effective reusable routines.

3 Meta-Verification

We formulate Meta-Verification as sequential decision-making under a resource budget. A Meta-Verifier selects verification actions based on accumulated evidence and remaining resources, terminating when it is ready to make a final decision. We first define this process, then introduce AgentVerificationBench to evaluate it, and finally describe how experience across tasks can improve the reusable harness supporting the agent.

3.1 Verification As A Sequential Policy

Verification tasks and actions. Let q be an underlying task and $t = (\tau_1, \ldots, \tau_N)$ a set of N candidate agent trajectories attempting to solve it. Each trajectory includes the recorded agent interaction and available task artifacts. Let $z_i \in \{0, 1\}$ indicate whether τ_i successfully solves q; these labels are hidden from the Meta-Verifier. Given q, t, available verification resources $\mathcal{R}$, and a nominal budget

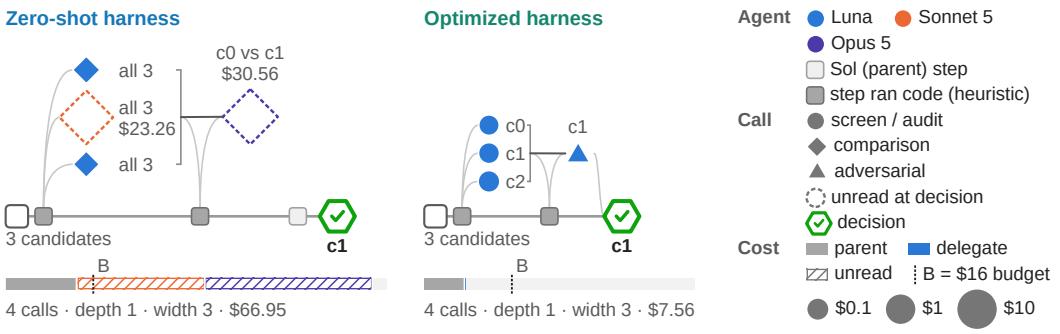

Figure 2: **One paired test task** (microarch-modeling). Both harnesses pick a passing candidate; the optimized harness costs 11% as much, and most zero-shot spend is unread at decision time.

$B > 0$, the Meta-Verifier gathers evidence and produces a terminal decision d. Under selection without rejection, $d \in \{1, \ldots, N\}$. When rejection is permitted, the decision space additionally includes $\perp$, denoting NONE. Selecting any successful candidate is correct, while $\perp$ is correct only if every candidate is unsuccessful. The formulation also accommodates pointwise verification by setting $N = 1$ and permitting rejection. The Meta-Verifier can inspect trajectories, tool outputs, and task artifacts (Zhuge et al., 2024); execute tests or other analysis programs (Han et al., 2025); and delegate verification to auxiliary models. Delegation options include pointwise assessment (Zheng et al., 2023) and comparison of candidates (Singh et al., 2026). The agent is free to choose any available verifier model, including one stronger than itself, subject to the associated cost. Its action space comprises these evidence-gathering operations and terminal actions that return a decision.

Sequential policy. Let $h_k = \{(a_\ell, o_\ell)\}_{\ell<k}$ denote the history before verification step k, where a_ℓ is a previous action and o_ℓ its observation. Let $C_{<k}$ be the total recorded verification cost incurred before that step, including the Meta-Verifier's own reasoning and delegated model calls. The remaining nominal budget is $b_k = B - C_{<k}$. The next action is selected according to

$$a_k \sim \pi_{\theta,H}\left(\cdot \mid q, t, \mathcal{R}, h_k, b_k\right), \tag{1}$$

where θ denotes the underlying model weights and H the reusable agent harness. If a_k gathers evidence, the Meta-Verifier observes its result, updates its history and remaining budget, and selects another action. If a_k is terminal, verification stops and the corresponding decision is returned. The policy jointly determines which evidence to gather, which verification mechanisms to use, how to allocate resources, and when to stop. The resulting action–observation sequence is the instance-specific verification procedure. Equation (1) describes the behavior induced by the agent and its harness; we do not train a separate policy network.

Execution modes. We compare two modes that differ in whether they permit the Meta-Verifier to revise its verification procedure during a task. In *Plan-then-Execute*, the agent first inspects the supplied task and trajectories and commits to a verification procedure before beginning executable checks or delegated calls. Conditional branches and stopping rules are permitted, but the agent is instructed not to replan during execution. *Adaptive Meta-Verification* permits the agent to reconsider its verification approach after observing intermediate evidence, changing its checks, model allocation, escalation decisions, or stopping rule. Thus, both modes allow responses to evidence, but only the Adaptive mode permits revision of the procedure itself. These modes are implemented through execution instructions, rather than externally registered and enforced plans. Section E provides their prompts.

3.2 AGENTVERIFICATIONBENCH

We introduce AgentVerificationBench (AVB) to evaluate verification decisions using fixed candidate sets, executable correctness labels, and controlled resource access. Table 1 summarizes both benchmark components.

Construction. For a generator configuration m and source task q, a verification instance is $t_{m,q} = (\tau_1, \ldots, \tau_N)$, consisting of N trajectories generated by m for q. Candidate correctness is determined

by the source executable graders. AVB-Code comprises 541 best-of-16 instances from Terminal-Bench 2.1 (Merrill et al., 2026) and DeepSWE (Huang et al., 2026). All retained instances contain both successful and unsuccessful candidates. AVB-Science comprises 189 best-of-3 instances from Terminal-Bench-Science (Terminal-Bench-Science Team, 2026), including instances in which all candidates are unsuccessful. Both components exclude incomplete candidate sets, unknown correctness labels, and all-correct instances; AVB-Code also excludes instances with observed hidden-test or reference-solution leakage.

Evaluation protocol. The verifier receives the source task, candidate trajectories, and available trace evidence and verification resources. Correctness labels, reference solutions, and private grader outcomes are withheld. Internet access is disabled during inference. AVB-Code requires selecting a successful trajectory among 16 candidates. AVB-Science requires selecting a successful trajectory among three candidates or returning NONE if all candidates fail. Invalid or missing decisions receive no accuracy credit. All instances sharing an underlying source task remain in the same data split, regardless of generator configuration. AVB-Code contains 217 training, 54 validation, and 270 test instances; AVB-Science contains 69, 42, and 78, respectively. The main code-domain comparisons use the common 98-instance test intersection. We evaluate selection accuracy, measured verification cost, and budget adherence. These metrics distinguish correct decisions from decisions whose verification cost exceeds the nominal budget. Dataset construction and exclusions are detailed in Section A; experimental resources and cost accounting are specified in Section 4.1.

3.3 Improving Meta-Verification Through Harness Optimization

While Meta-Verification adapts within a task, we investigate whether experience across tasks can improve its behavior without updating the model weights θ. We optimize the reusable harness H, which includes prompts, tools, skills, memory, and orchestration (Agrawal et al., 2026; Lee et al., 2026; Zhang et al., 2026b; Lin et al., 2026; Ye et al., 2026). Harness revisions can change how the agent selects verification actions, interprets evidence, allocates resources, and decides when to stop.

Cost-sensitive objective. Let D denote a batch of verification instances, each with a nominal budget $B_i > 0$. For harness H on instance i, let $y_i(H) \in \{0, 1\}$ indicate a valid, correct decision, and let $C_i(H)$ denote its total recorded verification cost. Costs include the Meta-Verifier and all auxiliary model calls, including metered failed or retried attempts. Invalid or missing decisions have $y_i(H) = 0$ and retain their incurred costs. We define per-instance utility and mean utility as

$$u_i^{\lambda}(H) = y_i(H) - \lambda \max\left(0, \frac{C_i(H)}{B_i} - 1\right), \qquad J_{\lambda}(H; D) = \frac{1}{|D|} \sum_{i \in D} u_i^{\lambda}(H), \tag{2}$$

where $\lambda \geq 0$ controls the penalty for budget overruns. Spending within budget incurs no penalty; exceeding the budget by a fraction r reduces utility by λr. We additionally report strict-budget utility,

$$u_i^{\text{strict}}(H) = y_i(H)\, \mathbf{1}\{C_i(H) \leq B_i\}. \tag{3}$$

Budgets are nominal rather than externally enforced: budget overruns ($C_i(H) > B_i$) are possible, and compliance is evaluated from the agent's realized cost. Section 4.1 specifies the experimental budgets and penalty scales.

Harness search and selection. Starting from an initial harness H_0, we use an iterative reflect–propose–evaluate procedure (Agrawal et al., 2026; Lee et al., 2026). At iteration k, a separate optimizer examines accumulated training records $\mathcal{E}_k$, including candidate trajectories, verification traces, check results, delegated responses, correctness labels, and costs. It proposes reusable harness revisions $\mathcal{H}_{k+1} = \mathcal{A}(H_0, \mathcal{E}_k)$, which are evaluated on common training task–budget pairs. Their outcomes augment the records used in subsequent iterations. Instance-specific answers and correctness labels must not be embedded in the reusable harness. Let $\mathcal{H}_{\text{eval}}$ contain the initial harness and all evaluated candidates. For the principal cost-sensitive search, we select

$$H^{\star} \in \arg\max_{H \in \mathcal{H}_{\text{eval}}} J_{\lambda}(H; D_{\text{val}}), \tag{4}$$

using a task-disjoint validation set. The selected harness is frozen before evaluation on the held-out test set. The optimizer receives training feedback but no validation or test feedback. The optimization settings and archived alternative selection protocol appear in Section F. The optimized harness is reused across tasks, while each task induces its own sequence of verification actions. Harness

optimization therefore changes the system supporting verification decisions across tasks; the resulting improvements need not arise specifically from stronger online replanning rather than more effective reusable routines.

4 EXPERIMENTS

We evaluate the sequential verification policy in Section 3 along three dimensions: selection accuracy, realized verification cost, and budget adherence. Our experiments ask whether online replanning improves verification decisions, whether experience across tasks improves the resource efficiency of a frozen-model Meta-Verifier, and how the resulting verification procedures change. These questions distinguish the choice of actions within a task from optimization of the reusable harness across tasks.

4.1 EXPERIMENTAL SETUP

Tasks and models. We use the evaluation protocol in Section 3.2. AVB-Code, built from Terminal-Bench 2.1 (Merrill et al., 2026) and DeepSWE (Huang et al., 2026), requires selecting a successful trajectory from 16 candidates. AVB-Science requires selecting a successful trajectory from three candidates or returning NONE when all three fail. The main comparisons use the 98 AVB-Code test tasks shared by all methods and all 78 AVB-Science test tasks. A separate comparison of execution modes uses 99 AVB-Code tasks. The source tasks in the training, validation, and test splits are disjoint (Appendix B). We use Codex (OpenAI, 2025) (Sol and Luna) and Claude Code (Anthropic, 2025) (Opus 4.8 and Opus 5) as Meta-Verifiers. Available verifier pools include Sol, Luna, Opus 5, Sonnet 5, and Haiku 4.5, with configuration-specific pools and harness restrictions detailed in Section E. We use the frozen token-price schedules in Section C. For AVB-Code, we compare Adaptive and Plan-then-Execute (Section 3.1) using Sol and Opus 4.8 at nominal budgets of \$0.50 and \$4. A larger comparison uses Luna, Sol, and Opus 5 at budgets of \$0.50, \$4, and \$16. AVB-Science uses Adaptive Sol with a nominal budget of \$16. Appendix A details the trajectory-generation configurations.

Table 1: AgentVerificationBench specifications.

Specification	AVB-Code	AVB-Science
Sources	TB 2.1; DeepSWE	TB-Science 0.1
Source tasks q	192	70
Generators m	4 per source	9
Instances $t_{m,q}$	541	189
Candidates N	16	3
Correct $\sum_i z_i$	1–15	0–2
Decision $d \in$	$\{1, \ldots, N\}$	$\{1, \ldots, N, \perp\}$
Train / val / main test	217/54/98	69/42/78

Comparison methods. We compare Meta-Verification with three ways of verifying candidates: LLM-as-a-Judge (Zheng et al., 2023), which scores each candidate from 1 to 20; V1 (Singh et al., 2026), which repeatedly compares pairs of candidates; and Agent-as-a-Verifier (AAV) (Zhuge et al., 2024), which directly inspects the candidates and makes a selection.

V1 is evaluated after N, $2N$, and $3N$ comparisons. We also test versions of LLM-as-a-Judge and V1 that use three task-specific evaluation criteria (Liu et al., 2023; Ye et al., 2024). For AVB-Science, we additionally allow AAV to continue verification using its previous transcript. This tests whether giving the same verifier more time produces similar benefits to harness optimization. Appendix B describes the evaluation subsets; Section 4.2 reports the model- and domain-specific coding comparisons.

Measures and cost accounting. Using the notation in Section 3.3, we report selection accuracy, mean verification cost, the fraction of tasks completed within budget, and the utility defined in Eq. (2). For reporting utility, we use $\lambda = 1$ for AVB-Code and $\lambda = 3.2$ for AVB-Science. At a \$16 budget, the science objective subtracts 0.2 utility for each dollar spent above budget. We also report strict-budget utility, which gives credit only when a decision is correct and its cost does not exceed the budget (Eq. (3)).

Verification cost includes all recorded model calls made by the Meta-Verifier and its auxiliary verifiers, including failed and retried calls. We use frozen, uncached token-price schedules and count reasoning tokens as output. The cost of searching for an improved harness is accounted for separately. Full pricing, the treatment of incomplete cost records, and differences between reported utility and execution instructions are given in Section C.

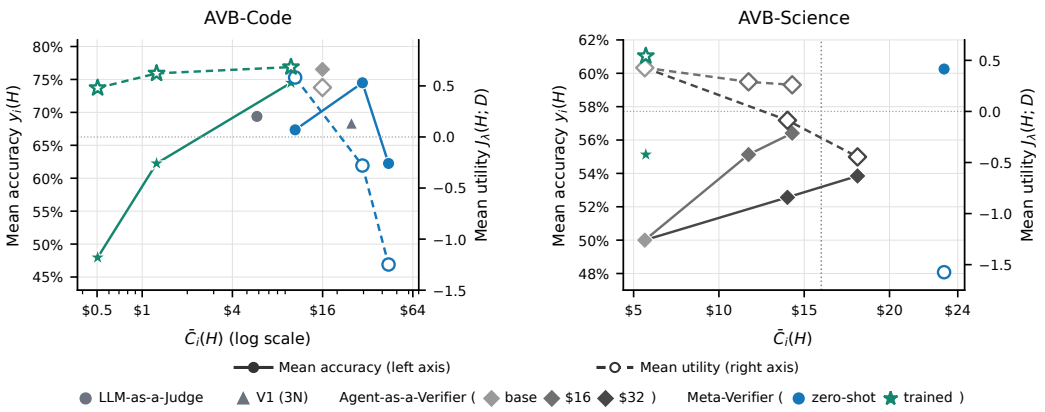

Figure 4: **Optimized harnesses attain the highest utility.** AVB-Code (left) and AVB-Science (right), with utility penalizing budget overruns (Eq. (2); $B_i = \$16$, $\lambda = 1$ and 3.2, respectively). Lines connect prompted budgets (AVB-Code) or AAV continuation checkpoints (AVB-Science).

4.2 Selection Accuracy And Online Replanning

Execution modes. Online replanning can improve candidate selection, but the gains depend on the model and budget. Figure 3 compares Adaptive Meta-Verification and Plan-then-Execute under strict-budget prompting on 99 best-of-16 AVB-Code instances, using Sol and Claude Code at $B \in \{\$0.5, \$4, \$16\}$. Adaptive Meta-Verification improves selection accuracy in five of the six model–budget settings and ties Plan-then-Execute in the remaining setting; the size of the gain varies by model and budget. The plots report selection accuracy: even a correct decision ($y_i(H) = 1$) receives zero strict-budget utility $u_i^{\text{strict}}(H)$ when $C_i(H) > B_i$.

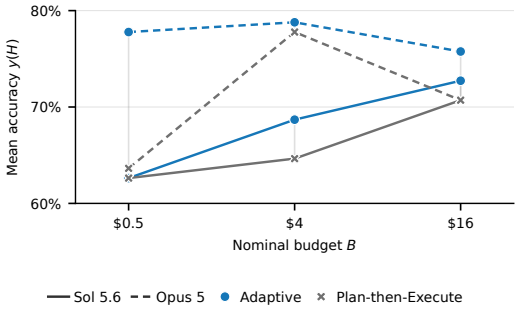

Figure 3: Online replanning improves selection accuracy.

Method comparison. Accuracy must be assessed at realized cost. Selection accuracy varies across domains and verification mechanisms, and more pairwise comparisons need not improve selection. Figure 4 compares LLM-as-a-Judge, V1 at $3N$, AAV, and Adaptive Meta-Verifiers with initial (H_0) and optimized harnesses on shared candidate sets, using Codex Sol throughout. Code results micro-average Terminal-Bench 2.1 and DeepSWE; Code utility is recomputed post hoc only for methods with exact per-instance agentic costs. At nominal $B = \$16$, the harness-optimized Sol Meta-Verifier selects correctly on 73/98 AVB-Code instances (74.5%) at a mean verification cost of $\$9.84$. In the initial $B = \$4$ comparison, Adaptive Sol reaches 73.7% mean selection accuracy at a mean cost of $\$18.96$. Across domains, Sol V1 reaches 94.1% on Terminal-Bench 2.1, whereas LLM-as-a-Judge outperforms V1 on DeepSWE (64.6% versus 55.4%).

Budget adherence. Initial harnesses provide poor budget adherence. Constructing instance-specific verification procedures does not by itself ensure control of cumulative cost. Figure 6 reports recorded costs and budget adherence for the $B = \$16$ cohorts. At $B \in \{\$0.50, \$4\}$, only 0–1% and 9.1–22.2% of initial runs, respectively, satisfy $C_i(H_0) \leq B_i$; median cost exceeds the nominal budget in every condition. At $B = \$16$, Sol with H_0 incurs median costs of $\$38.83$ on AVB-Code and $\$20.43$ on AVB-Science, with budget adherence of 24/98 and 33/78, respectively. These results motivate improving not only which verification actions an agent chooses, but also how much it spends.

4.3 Improving Resource Efficiency Through Harness Optimization

Harness selection. To address poor budget adherence, we optimize the reusable harness using experience from training tasks while keeping model weights θ fixed. Figure 5 details the AVB-

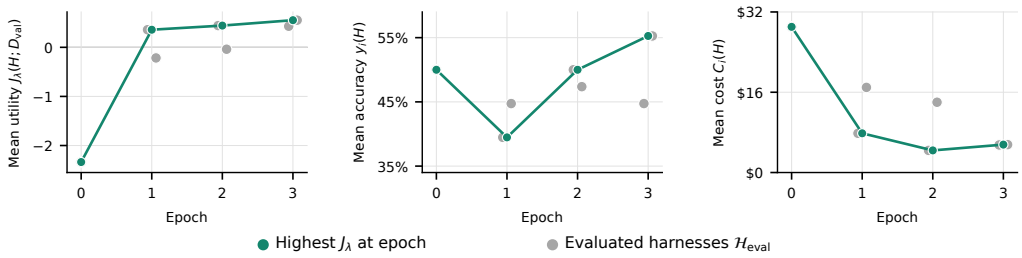

Figure 5: **Harness selection improves utility while reducing cost.** AVB-Science results on validation instances with exact costs. Panels show mean utility, selection accuracy, and verification cost over epochs, with utility subtracting a budget-overrun penalty from accuracy (Eq. (2); $B_i = \$16$, $\lambda = 3.2$).

Science validation trajectory. Relative to H_0, the selected harness $H^\star = H_{3c1}$ raises $J_\lambda(H; D_{\text{val}})$ from -2.337 to 0.553 and selection accuracy from 50.0% to 55.3%, while reducing mean verification cost from $\$29.04$ to $\$5.56$. Following Section 3.3, a separate Sol optimizer $\mathcal{A}$ proposes two harnesses per epoch for three epochs, evaluating each on 17 scheduled training instances and receiving no validation or test feedback. Training guides search; we select $H^\star$ from H_0 and the six proposals by $J_\lambda(H; D_{\text{val}})$ on the common 38-instance validation subset with exact cost records, using $B_i = \$16$ and $\lambda = 3.2$. We freeze $H^\star$ before evaluating transfer to all 78 held-out test instances.

Test performance. The held-out gain is in resource efficiency. Figure 4 compares the initial and optimized Meta-Verifiers on AVB-Science test instances. Relative to H_0, $H^\star$ reduces mean recorded test cost from $\$23.20$ to $\$5.72$ and increases budget adherence from 33/78 (42.3%) to 77/78 (98.7%), while selection accuracy decreases from 60.3% to 55.1%. $H^\star$ is cheaper on 70/78 paired instances, with median cost $\$5.43$; the accuracy comparison has exact McNemar $p = 0.45$. These results establish improved cost control without establishing an accuracy improvement or equivalence. For H_0, 72/78 instances have exact final costs; the remaining six have lower bounds already exceeding B_i. Its reported mean cost is therefore a lower bound, while budget-adherence classifications remain unchanged. The two science conditions also differ in cost-penalty instructions and concurrency limits, so this comparison evaluates the resulting verification setups without isolating individual harness changes. Figure 2 shows one paired instance.

Continuation baselines. Reusable harness revisions improve verification efficiency beyond the evaluated trajectory continuations. Figure 4 compares reusable harness revisions with transcript-conditioned continuation on the same 78 AVB-Science instances. Luna and Sol AAV both attain 50.0% selection accuracy; initial Sol AAV costs $\$5.66$ per instance. Under the $\$16$ continuation schedule, one and two additional passes reach 55.1% and 56.4% selection accuracy at mean cumulative costs of $\$11.74$ and $\$14.30$. The $\$32$ schedule yields 52.6% and 53.8% at $\$14.02$ and $\$18.12$. The Meta-Verifier with $H^\star$ achieves 55.1% selection accuracy at $\$5.72$ and the highest mean utility among these conditions, $J_\lambda(H^\star; D) = 0.542$ for the test set D. This matches one continuation's observed accuracy at less than half its mean cost. Separate 54-instance validation experiments also improve budget adherence under an archived accuracy-first selection rule; Appendix F reports these results.

4.4 Verification Procedures And Resource Allocation

Resource efficiency. Harness optimization reduces verification cost and improves budget adherence in both domains. Figure 6 compares the initial and optimized Sol Meta-Verifiers at $B = \$16$. On the 98 coding tasks, median cost falls from $\$38.8$ to $\$8.8$, while budget adherence rises from 24/98 (24.5%) to 82/98 (83.7%). On the 78 science tasks, median cost falls from $\$20.4$ to $\$5.4$, while budget adherence rises from 33/78 (42.3%) to 77/78 (98.7%). These results use runtime accounting and do not depend on a model's description of its own verification behavior.

Procedure structure and cost. The trace analysis points to simpler, cheaper verification routines after science harness optimization. An independent GPT-5.5 analyzer reviews 254 executed Sol traces: 98 zero-shot coding instances, 78 zero-shot science instances, and the same 78 science instances under the frozen optimized harness. It receives each task, verification prompt, and parent trace, including delegated responses, without ground-truth correctness labels. Its technique labels describe observed behavior and do not measure correctness. Pairwise comparison appears in 91.8% of zero-shot coding

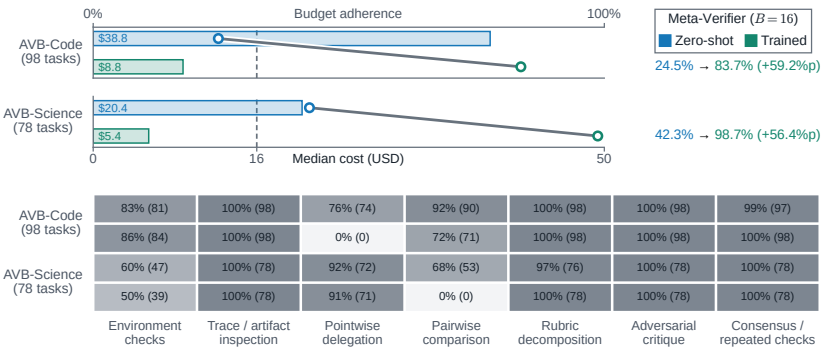

Figure 6: **Harness optimization lowers cost and improves budget adherence.** Sol at $B = \$16$. Upper panels overlay median-cost bars and budget-adherence lines . Lower cells show technique prevalence as percentage (task count); labels are multi-valued.

traces and 67.9% of zero-shot science traces, but in none of the trained science traces; pointwise delegation appears in 91.0% of the latter. The optimized harness standardizes the science procedure: 59/78 instances use three parallel Luna audits followed by one further call, and the harness no longer delegates to the more expensive verifier models used initially. Reduced delegate spending accounts for 64% of the mean cost reduction, while the parent accounts for 95.7% of the optimized harness's remaining cost.

The trained Code technique labels in Fig. 6 come from a separate evidence-linked review and are provisional pending annotation with the original analyzer; we do not use them to draw conclusions about changes in coding strategies. Together, the execution records and measured costs support learning more economical verification procedures through experience. They do not isolate improved online replanning from adopting inexpensive checks, delegation, and stopping routines that can be reused across tasks. Appendix D shows example procedures.

Limitations. Results on fixed coding and science candidate sets do not establish performance on single trajectories or other agent types, and executable graders may miss aspects of task success. Execution modes rely on instructions, not enforced plans; Plan-then-Execute permits conditional branches. Science harnesses also change penalty instructions and concurrency limits, preventing isolation of replanning or individual harness effects. Lower held-out science cost establishes neither improved nor equivalent accuracy. Six incomplete initial science cost records make the initial mean cost a lower bound; costs cover model tokens, with harness-search costs reported separately.

5 Conclusion

We study Meta-Verification as the sequential selection of verification actions based on accumulated evidence and remaining resources. Rather than applying a prescribed procedure, a Meta-Verifier can choose which evidence to gather, which checks to perform, when to consult other models, and when to stop. We introduce AgentVerificationBench to evaluate this capability across coding and scientific agent trajectories. Our experiments show that agents construct different verification procedures for different tasks, but online replanning does not consistently improve selection accuracy across models. Unmodified Meta-Verifiers also frequently exceed their stated budgets. We further show that experience can improve verification resource efficiency by optimizing the reusable agent harness while keeping model weights fixed. On held-out science tasks, the optimized verifier substantially reduces cost and improves budget adherence, although its selection accuracy is lower than that of the more expensive initial verifier. These findings highlight the importance of evaluating verification accuracy and resource use together. They also motivate further work on identifying which verification decisions benefit from online replanning and how agents can reliably stay within their resource budgets.

AI USE STATEMENT

Beyond the LLM use within the methods and experiments described in this paper, we used LLMs only as writing aids to improve the clarity and readability of the manuscript. We did not otherwise use LLMs to generate synthetic datasets; develop theoretical models or conceptual frameworks; formulate mathematical claims, provide critical ingredients for proofs, or assist in writing proofs; propose or refine hypotheses; design or provide feedback on research methodology or experiments; implement methods; assist with translation; clean or reformat datasets; support qualitative or thematic data analysis; or interpret results. We reviewed all AI-assisted text and take responsibility for the final content of this work, including its text, claims, and artifacts.

REFERENCES

Pranjal Aggarwal, Aman Madaan, Ankit Anand, Srividya Pranavi Potharaju, Swaroop Mishra, Pei Zhou, Aditya Gupta, Dheeraj Rajagopal, Karthik Kappaganthu, Yiming Yang, et al. Automix: Automatically mixing language models. *Advances in Neural Information Processing Systems*, 37: 131000–131034, 2024.

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. In *International Conference on Learning Representations*, volume 2026, pp. 8479–8565, 2026.

Anthropic. Claude code: An agentic coding tool., 2025. URL https://claude.com/product/claude-code/.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024. URL https://arxiv.org/abs/2407.21787.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021. URL https://arxiv.org/abs/2110.14168.

Google. gemini-cli., 2026. URL https://github.com/google-gemini/gemini-cli/.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yujiu Yang, Nan Duan, Weizhu Chen, et al. Critic: Large language models can self-correct with tool-interactive critiquing. In *International Conference on Learning Representations*, volume 2024, pp. 57734–57811, 2024.

Jiuzhou Han, Wray L Buntine, and Ehsan Shareghi. Verifiagent: a unified verification agent in language model reasoning. In *EMNLP (Findings)*, pp. 16410–16431, 2025.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *International Conference on Learning Representations*, volume 2025, pp. 21344–21377, 2025.

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.

Seungone Kim, Juyoung Suk, Shayne Longpre, Bill Yuchen Lin, Jamin Shin, Sean Welleck, Graham Neubig, Moontae Lee, Kyungjae Lee, and Minjoon Seo. Prometheus 2: An open source language model specialized in evaluating other language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 4334–4353, 2024.

Jacky Kwok, Shulu Li, Pranav Atreya, Yuejiang Liu, Yixing Jiang, Chelsea Finn, Marco Pavone, Ion Stoica, and Azalia Mirhoseini. Llm-as-a-verifier: A general-purpose verification framework. *arXiv preprint arXiv:2607.05391*, 2026a.

Jacky Kwok, Xilun Zhang, Mengdi Xu, Yuejiang Liu, Azalia Mirhoseini, Chelsea Finn, and Marco Pavone. Scaling verification can be more effective than scaling policy learning for vision-language-action alignment. In *European Conference on Computer Vision*, pp. 37–55. Springer, 2026b.

Yoonho Lee, Roshen Sanjay Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses. In *Third Conference on Language Modeling*, 2026. URL https://openreview.net/forum?id=tmbOUyFx3R.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *International Conference on Learning Representations*, volume 2024, pp. 39578–39601, 2024.

Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Zhiheng Xi, Xuanjing Huang, Hang Yan, Zhenhua Han, Tao Gui, and Yu-Gang Jiang. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses. *arXiv preprint arXiv:2604.25850*, 2026. URL https://arxiv.org/abs/2604.25850v4.

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. G-eval: Nlg evaluation using gpt-4 with better human alignment. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 2511–2522, 2023.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in neural information processing systems*, 36:46534–46594, 2023.

Mike A Merrill, Alexander Glenn Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Kumar Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Kwesi Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Jenia Jitsev, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=a7Qa4CcHak.

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

Sungho Park, Wonjoong Kim, Rongyuan Tan, Jue Zhang, Wook-Shin Han, Pengfei Gao, Chanyoung Park, Yongqiang Yao, Rao Fu, Elsie Nallipogu, et al. Autosaddler: Automatic harness optimization with durable updates from agent execution traces. *arXiv preprint arXiv:2608.23041*, 2026.

Corby Rosset, Pratyusha Sharma, Andrew Zhao, Miguel Gonzalez-Fernandez, and Ahmed Awadallah. The art of building verifiers for computer use agents. *arXiv preprint arXiv:2604.06240*, 2026.

Jon Saad-Falcon, E Kelly Buchanan, Mayee F Chen, Tzu-Heng Huang, Brendan McLaughlin, Tanvir Bhathal, Shang Zhu, Ben Athiwaratkun, Frederic Sala, Scott Linderman, et al. Shrinking the generation-verification gap with weak verifiers. *arXiv preprint arXiv:2506.18203*, 2025.

Amrith Setlur, Nived Rajaraman, Sergey Levine, and Aviral Kumar. Scaling test-time compute without verification or rl is suboptimal. *arXiv preprint arXiv:2502.12118*, 2025.

Harman Singh, Xiuyu Li, Kusha Sareen, Monishwaran Maheswaran, Sijun Tan, Xiaoxia Wu, Junxiong Wang, Alpay Ariyak, Qingyang Wu, Samir Khaki, et al. v_1: Unifying generation and self-verification for parallel reasoners. *arXiv preprint arXiv:2603.04304*, 2026.

Charlie Victor Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling parameters for reasoning. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=4FWAwZtd2n.

Terminal-Bench-Science Team. Terminal-Bench-Science: Evaluating AI agents on research workflows across scientific domains, 2026. URL https://github.com/harbor-framework/terminal-bench-science.

Jonathan Uesato, Nate Kushman, Ramana Kumar, Francis Song, Noah Siegel, Lisa Wang, Antonia Creswell, Geoffrey Irving, and Irina Higgins. Solving math word problems with process-and outcome-based feedback. *arXiv preprint arXiv:2211.14275*, 2022.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

Yixuan Weng, Minjun Zhu, Fei Xia, Bin Li, Shizhu He, Shengping Liu, Bin Sun, Kang Liu, and Jun Zhao. Large language models are better reasoners with self-verification. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 2550–2575, 2023.

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

Haoran Ye, Xuning He, Vincent Arak, Haonan Dong, and Guojie Song. Meta context engineering via agentic skill evolution. *arXiv preprint arXiv:2601.21557*, 2026.

Seonghyeon Ye, Doyoung Kim, Sungdong Kim, Hyeonbin Hwang, Seungone Kim, Yongrae Jo, James Thorne, Juho Kim, and Minjoon Seo. Flask: Fine-grained language model evaluation based on alignment skill sets. In *International Conference on Learning Representations*, volume 2024, pp. 55361–55414, 2024.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Tjarko Lange, and Jeff Clune. Darwin gödel machine: Open-ended evolution of self-improving agents. In *The Fourteenth International Conference on Learning Representations*, 2026a. URL https://openreview.net/forum?id=pUpzQZTvGY.

Lunjun Zhang, Arian Hosseini, Hritik Bansal, Seyed Mehran Kazemi, Aviral Kumar, and Rishabh Agarwal. Generative verifiers: Reward modeling as next-token prediction. In *International Conference on Learning Representations*, volume 2025, pp. 12476–12505, 2025.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic context engineering: Evolving contexts for self-improving language models. In *The Fourteenth International Conference on Learning Representations*, 2026b. URL https://openreview.net/forum?id=eC4ygDs02R.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems*, 36:46595–46623, 2023.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406*, 2023.

Lianghui Zhu, Xinggang Wang, and Xinlong Wang. Judgelm: Fine-tuned large language models are scalable judges. In *International Conference on Learning Representations*, volume 2025, pp. 51257–51296, 2025.

Mingchen Zhuge, Changsheng Zhao, Dylan R. Ashley, Wenyi Wang, Dmitrii Khizbullin, Yunyang Xiong, Zechun Liu, Ernie Chang, Raghuraman Krishnamoorthi, Yuandong Tian, Yangyang Shi, Vikas Chandra, and Jürgen Schmidhuber. Agent-as-a-judge: Evaluate agents with agents. *arXiv preprint arXiv:2410.10934*, 2024. URL https://arxiv.org/abs/2410.10934.

Appendix

A Benchmark Composition and Splits **14**

B Evaluation Cohorts **17**

C Verification Cost Accounting **18**

D Example Verification Procedures **19**

D.1 Zero-Shot Examples 19

D.2 What Harness Optimization Learned 19

E Meta-Verifier Inference Prompts **22**

E.1 Shared Prompt 22

E.2 Mode-Specific Instructions 28

F Harness Optimization Details **29**

F.1 Search And Selection 29

F.2 Results And Cost Accounting 29

G Responsible-Use Statement **29**

A BENCHMARK COMPOSITION AND SPLITS

Collection and completeness. The frozen collection contains 88 Terminal-Bench 2.1 source questions and 113 DeepSWE v1.1 source questions. Both domains use GPT-5.4-mini, Luna, and Gemini 3.7 Flash; the fourth generator is Claude Haiku 4.5 for TB2 and Claude Sonnet 4.6 for DSWE. The four-model collection contains 804 verification tasks $t_{m,q}$ and 12,864 rollouts, with 16 rollouts per task. We retain reasoning, tool calls, tool outputs, and final responses, including unsuccessful, timed-out, and failed runs. Each source question's executable outcome grader provides correctness labels. Of these rollouts, 12,815 have known labels and 49 have no verdict, for example because the outcome grader timed out or the run produced no work. A task is complete only if all 16 rollouts are present and labeled. At the unbounded cutoff, 789 tasks are complete: 133 all-wrong, 102 all-correct, and 554 swing (Table 2). All-wrong tasks contain no correct rollouts, all-correct tasks contain only correct rollouts, and swing tasks contain both correct and incorrect rollouts.

Table 2: AgentVerificationBench composition by domain.

	Collection		Complete tasks				Evaluation
Benchmark	Questions	Rollouts	All-wrong	All-correct	Swing	Total	Clean swing
TB2	88	5,632	52	89	205	346	192
DSWE	113	7,232	81	13	349	443	349

Full export inventory. Table 3 also includes 113 auxiliary DSWE/Claude Haiku 4.5 tasks with one rollout each: one correct, 108 incorrect, and four unknown. These account for the difference between the 804 tasks in the four-model collection and the 917 tasks in the full export. They do not contribute to the complete 16-trajectory tasks in Table 2. Known-label counts include labeled rollouts from incomplete tasks.

Leakage exclusions. The corpus records both the executable grader's outcome and whether a trace contains evidence of obtaining hidden tests or a reference solution. Such evidence occurs in 66 rollouts across 15 complete TB2 tasks: two all-correct and 13 swing tasks. These tasks remain in the corpus, but the 13 affected swing tasks are excluded from best-of-N evaluation, leaving 192 TB2 and 349 DSWE tasks over 192 distinct source questions. These exclusions address observed leakage; they do not establish the absence of pretraining memorization.

Task-disjoint splits. All verification tasks $t_{m,q}$ sharing a source question q are assigned to the same train, validation, or test split across generators m. The assignment is stratified by domain, generator, and the number of correct rollouts. Nine collected source questions have no clean swing task and are unassigned. Table 4 gives the frozen membership counts. The main comparisons use the 98-task intersection shared by all methods; the evaluation cohorts below document the original 100-task subset and its exclusions.

Verification protocol.

Correctness labels, rewards, grader outputs, task statistics, and reference solutions are withheld from verifiers; labels remain in a separate scoring store. Internet access is disabled during inference. Each input is content-addressed so the exact evaluation can be reconstructed. Harness search uses only the training split, the validation split is used to select the final harness, and the test split is accessed once for reporting. Best-of-N evaluation ($N = 16$) reports pick-correct accuracy, the fraction of evaluated swing tasks for which the selected rollout is correct, at fixed comparison budgets.

Evaluation cohorts. The original AVB-Code evaluation uses a 100-task hard test set: 35 Terminal-Bench 2.1 and 65 DeepSWE tasks. The three-model evaluation excludes the prompt-development task, leaving 99 tasks. Comparisons shared across methods use the exact 98-task intersection: 33 Terminal-Bench 2 and 65 DeepSWE tasks. Trained Meta-Verifier results also include a separate collaborator-reported 100-task held-out snapshot; the updated trained Sol $B = \$16$ point uses the audited 98-task intersection. AVB-Science uses its own 78-task test split and selection-or-NONE protocol described below. Results retain their respective cohort denominators and are reported separately.

AVB-Science. We construct AVB-Science from Terminal-Bench-Science 0.1 trajectories covering 70 source questions across earth, life, mathematical, and physical sciences and engineering. The

Table 3: Detailed corpus inventory at the unbounded cutoff. Full-export counts include the auxiliary single-rollout DSWE tasks. "Clean" excludes tasks with detected hidden-test or reference-answer leakage.

Component	TB2	DSWE	Total
Source questions	88	113	201
Verification tasks	352	565	917
Complete 16-trajectory tasks	346	443	789
Known-label trajectories	5,610	7,314	12,924
Correct trajectories	3,274	2,509	5,783
Incorrect trajectories	2,336	4,805	7,141
Unknown-label rollouts	22	31	53
Swing tasks before leakage exclusions	205	349	554
Swing tasks excluded for leakage	13	0	13
Clean swing tasks	192	349	541

finalized release contains 189 complete tasks and 567 original trajectories from nine generator models; each task $t_{m,q}$ contains three rollouts from the same model m on the same source question q. The generators are Sol, Luna, and GPT-5.6 Terra; Claude Opus 4.8 and Claude Opus 5; and Fable 5, GLM 5.3, Grok 4.6, and Kimi K3. Based on source executable-grader labels, 59 tasks have zero, 86 have one, and 44 have two correct candidates. All-correct tasks, unknown-label triples, and three tasks with unavailable original transcripts are excluded. The verifier selects a successful trajectory or returns NONE when all three candidates have failed; either correct candidate receives credit in a two-pass task. The dataset has its own training, validation, and test splits, disjoint in source questions, with 69, 42, and 78 verification tasks over 25, 15, and 30 source questions, respectively. Split construction approximately stratifies generator-by-outcome classes and scientific domains while preserving source labels and candidate ordering. The expected accuracies of always choosing NONE and uniformly selecting a candidate differ by less than one percentage point in each split. Verifier inputs expose only the source question and its three candidate trajectories; private labels, source grading results, selection plans, and release-wide statistics are withheld, and explicit generator metadata is removed from the input traces. These labels measure success under the benchmark graders.

Table 4: Clean swing-task splits. Train, validation, and test are disjoint in source questions.

Split	TB2 tasks	DSWE tasks	Total tasks	Questions
Train	77	140	217	71
Validation	20	34	54	21
Test	95	175	270	100

B EVALUATION COHORTS

Table 5 distinguishes the evaluation subsets used in Section 4. Counts refer to verification tasks $t_{m,q}$, each containing N candidate trajectories, rather than distinct source questions q. AVB-Code uses $N = 16$ and AVB-Science uses $N = 3$. Results retain their respective cohort denominators and are reported separately.

The full benchmark training/validation/test splits contain 217/54/270 AVB-Code tasks and 69/42/78 AVB-Science tasks, with source questions disjoint across splits (Section A). The 98-task AVB-Code cohort used for comparisons shared across methods is a subset of the 270-task test split, while the AVB-Science evaluation uses all 78 test tasks. The "Train / val / main test" row in Table 1 combines the full training and validation splits with the main evaluated test cohort. Its 217/54/98 coding counts therefore use 98 for the shared evaluation subset; the full coding test split contains 270 tasks. For AVB-Science, the main evaluated cohort and full test split both contain 78 tasks.

Table 5: Evaluation cohorts and reporting locations. All rows describe test tasks drawn from splits disjoint in source questions.

Domain	Comparison	Tasks	Composition and selection	Reported in
AVB-Code	Main comparison	98	Intersection shared by all methods, with 33 Terminal-Bench 2.1 and 65 DeepSWE tasks.	Figure 4
AVB-Code	Execution modes	99	Original 100-task subset after excluding the prompt-development task.	Figure 3
AVB-Code	Broader baselines	100	Original hard test subset, with 35 Terminal-Bench 2.1 and 65 DeepSWE tasks.	Section 4.2
AVB-Science	Main comparison and continuation	78	Full science test split over 30 source questions. The protocol permits $d = \perp$ when all $z_i = 0$.	Figure 4 and Section 4.3

Earlier trained Meta-Verifier results include a separate collaborator-reported 100-task held-out snapshot. The updated trained Sol result at $B = \$16$ uses the audited 98-task intersection in Table 5. Harness-selection validation subsets are described in Sections 4.3 and F.

C VERIFICATION COST ACCOUNTING

We measure $C_i(H)$ in normalized US dollars by summing input and output charges across the Meta-Verifier and its auxiliary calls, including rubric generation when used. Cached input and cache writes are charged at the uncached input rate; reasoning tokens are included in output. Budgets and budget adherence use the same accounting. Outer-loop harness optimization costs are accounted for separately.

Table 6: Frozen normalized token prices in US dollars per million tokens. Long-context rates apply when a request exceeds 272,000 input tokens. A dash indicates that the frozen catalog specifies no separate long-context tier.

	Standard context		Long context	
Model	Input	Output	Input	Output
Sol	4.00	20.00	8.00	30.00
Luna	0.20	1.20	0.40	1.80
Opus 4.8	5.00	25.00	—	—
Opus 5	5.00	25.00	—	—
Sonnet 5	2.00	10.00	—	—
Haiku 4.5	1.00	5.00	—	—
GPT-5.6 Terra (earlier pool)	2.00	12.00	4.00	18.00
Sonnet 5 (earlier catalog)	3.00	15.00	—	—

Each run retains its frozen rate card, including any context-length tiers. Frozen schedules may differ between run catalogs, and a tiered model's effective input and output prices depend on the request's context length. The archived six-agent harness prompt includes GPT-5.6 Terra and charges the earlier Sonnet 5 configuration at \$3/\$15 per million input/output tokens; the later catalog charges \$2/\$10. For tiered accounting, the request's input length selects both its input and output rates. Historical aggregate records without sufficient request-level information retain their recorded costs and cannot establish exact long-context charges.

Costs are averaged per task; continuation costs are cumulative. Missing provider usage is not imputed, and incomplete cost records are identified as lower bounds where applicable. All scheduled tasks remain in the accuracy denominator, including invalid or missing decisions. Section F details recorded zero-cost entries and incomplete telemetry in the archived harness results.

Reported utility and execution instructions. The main method comparison reports utility under a common budget of $B = \$16$, using $\lambda = 1$ for AVB-Code and $\lambda = 3.2$ for AVB-Science in Eq. (2). For Science, this is equivalent to subtracting 0.2 utility for each dollar above budget. Code utility is recomputed post hoc for methods with exact per-instance agentic costs. These reporting settings are distinct from the instructions used to execute each run. The initial Science prompt used $u = y - \max(0, C/B - 1)$ and permitted six concurrent delegated calls. The optimized Science prompt used $u = y - 0.2\max(0, C - 16)$ and permitted three. The resulting comparison combines changes in harness, penalty instructions, and concurrency; it does not isolate any one change (Section D.2).

D EXAMPLE VERIFICATION PROCEDURES

This appendix shows how Sol Meta-Verifiers checked candidates on single tasks. All runs use the adaptive mode with the soft-overrun objective at $B = \$16$. The examples come from zero-shot AVB-Code (98 test tasks, 16 candidates each), zero-shot AVB-Science (78 held-out test tasks, 3 candidates plus NONE), and the same 78 science tasks under the optimized harness H_{3c1}. We picked the examples by hand to show different procedures and to include failures. They are case studies, not a representative sample; Section 4.4 gives statistics over whole runs.

Reading the graphs. Each panel is one task, rebuilt from its saved traces and cost report. The task and its candidates are on the left; the decision is the hexagon on the right (green check: correct; red cross: wrong). A decision is correct if the grader passes the chosen candidate, or if it is NONE and no candidate passes; invalid decisions count as wrong. Squares are the parent's own work, such as running candidate code. Every other node is one delegated call: colour gives the model, shape the role (screen or audit, comparison, or adversarial check), and size the cost. A dashed outline means the result was not read before the decision; a cross means the call failed. Lines link each call to the earlier results it builds on. Depth is the longest chain of dependent calls, counted from 0; width is the most calls at one depth. The bar below splits the total cost into parent and delegates, with unread calls hatched and a dotted line at B. An asterisk marks a cost lower bound where one call was unmetered. Candidates c0, c1, … are numbered in the shuffled order the Meta-Verifier saw. Call roles, read detection, and the "step ran code" marker are heuristics computed from the traces.

D.1 ZERO-SHOT EXAMPLES

Figures 7 and 8 show four tasks per domain; the captions describe each one. The procedure changes from task to task. The parent's own checks often decide (Fig. 7a,b and Fig. 8b), and in two examples it correctly overrides its delegates (Fig. 7b and Fig. 8b), although overrides are not always correct. The main weakness is cost control, not building the procedure. Costly calls are launched after cheap evidence already agrees, or are left unread at the decision (Fig. 7c,d and Fig. 2). In AVB-Code, only 24/98 tasks stayed within B, and the parent alone exceeded B in 49/98. In 60/98 tasks, at least one call (138 calls, \$1,536 in total) shows no sign of being read before the decision (a heuristic check, not proof). In AVB-Science, the parent alone exceeded B in 23/78 tasks, and a correct NONE came in only 9 of the 25 tasks where no candidate passes.

D.2 WHAT HARNESS OPTIMIZATION LEARNED

A separate Sol optimizer reviewed how the current harnesses did on training tasks and proposed two new harnesses per epoch for three epochs. The frozen result is H_{3c1} (Fig. 5). The seed harness h_0 (113 lines), used in the zero-shot AVB-Science run, offers its methods as "*optional building blocks, not a required algorithm*" and asks delegates for 1–20 scores. H_{3c1} (146 lines) replaces this with a fixed recipe: map the task's pass requirements ("gates"), check the candidates once directly, run one parallel panel of up to three Luna calls, make at most one extra call, and decide with a four-row table (each candidate passes, or all three fail). Its settings cap the parent at 28 turns and three concurrent calls (zero-shot allowed six). Table 7 lists the main rules and what changed on the 78 test tasks. The trained run is much cheaper (mean \$5.72 versus \$23.20 per task; the zero-shot mean is a lower bound) but less accurate: 43/78 correct (55.1%) versus 47/78 (60.3%) (Fig. 4); the difference is not significant (exact McNemar $p = 0.45$). The runs also differ in objective text and concurrency, so the table shows how behaviour changed, not a controlled harness effect. No trained AVB-Code traces were available, so this subsection covers AVB-Science only.

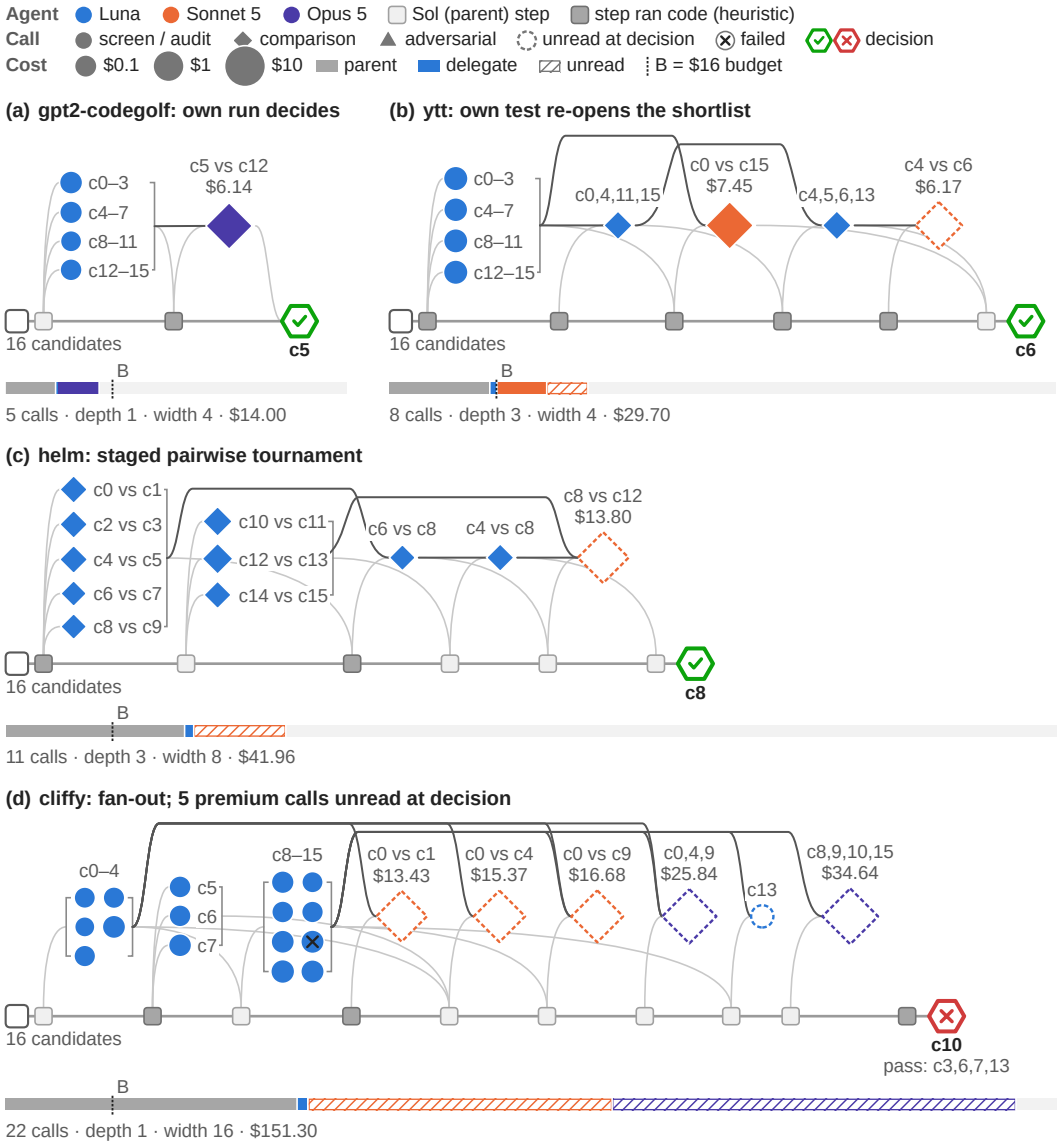

Figure 7: **Zero-shot coding procedures range from targeted checks to costly fan-out.** AVB-Code, Sol, $B = \$16$, 16 candidates. (a) Cheap screens reject all 16; the parent's own runs and one Opus 5 comparison pick c5 (correct, \$14.00). (b) The parent's own test reopens the shortlist; it picks c6, not Sonnet 5's c15 (correct, \$29.70). (c) A Luna tournament picks c8; a \$13.80 Sonnet 5 check is read only after the decision (correct, \$41.96). (d) A 22-call fan-out picks failing c10; six calls (\$106.06) are unread at the decision (total \$151.30).

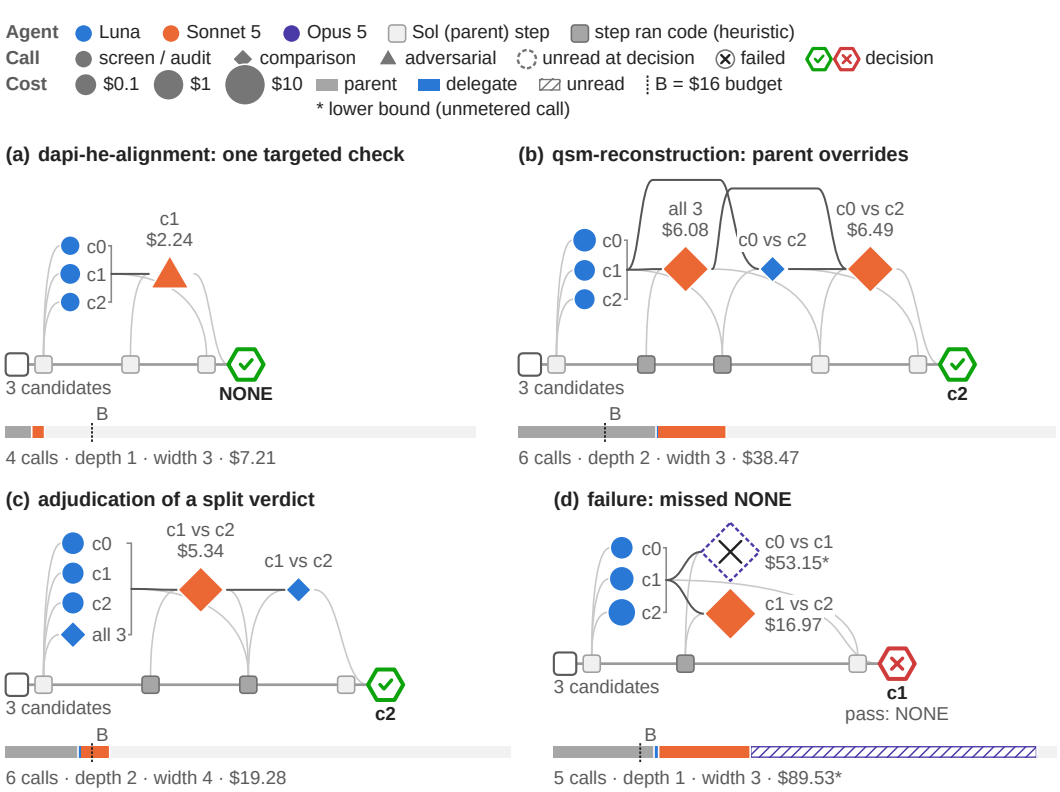

Figure 8: **Zero-shot science procedures can settle disagreements but can also mistake validity for correctness.** AVB-Science, Sol, $B = \$16$, 3 candidates plus NONE. (a) Cheap audits fail all three; one Sonnet 5 check ($2.24) confirms c1 fails; correct NONE ($7.21). (b) Two Sonnet 5 reviews favour c0; the parent overrides them and picks c2, the only passing candidate ($38.47). (c) Cheap checks disagree; one Sonnet 5 call ($5.34) fails c1 and passes c2, the correct pick ($19.28). (d) The parent takes structural validity for correctness and picks c1, but no candidate passes ($89.53*; *cost is a lower bound, one failed call was unmetered).

Table 7: Main rules learned by harness optimization on AVB-Science, paraphrased from the frozen H_{3c1} prompt (quotes are exact). The right column compares the trained (TR) and zero-shot (ZS) runs on the 78 test tasks; (regex) and (heuristic) mark heuristic counts.

Rule in H_{3c1}	Observed on test
Only Luna delegates. Up to three parallel Luna calls; Opus 5, Sonnet 5, and delegated Sol calls are forbidden.	TR: all 272 calls go to Luna ($19.24 in total). ZS: 110 Opus 5, Sonnet 5, and delegated Sol calls cost $873.84 of $892.54 in delegate spend.
One gate per delegate, facts only. Give each delegate only its candidate and its highest-risk scored gate; "*Ask for facts, not numerical scores or a generic recommendation.*"	Calls asking for a 1–20 score: TR 0/272, ZS 309/310. TR ran one parallel one-candidate panel in 71/78 tasks; the other 7 made no calls.
One extra call, planned first. At most one fourth Luna call, only after the first evidence is combined; before it, write the two outcomes that would lead to different final actions.	TR: 59/78 tasks made a fourth call, each after all three panel replies and on one candidate.
No voting. "*Do not vote over child verdicts.*" Fix a provisional action before reading the panel; change it only for validated facts.	The word "provisional" appears in parent messages in 67/78 TR and 6/78 ZS tasks (regex).
High bar for NONE. Return NONE only if all three candidates have a demonstrated, material failure case and the weakest case survives a rescue attempt; uncertainty alone is not enough.	Expected fewer false NONEs; not seen. Both runs return 11 NONEs (TR 8 correct, ZS 9). Correct NONE when no candidate passes: TR 8/25, ZS 9/25.
All-fail stays open. Relative rank changes which candidate leads, but it is not evidence against the all-fail event.	An explicit all-fail row appears in parent messages in 55/78 TR and 0/78 ZS tasks (regex).
Budget. Aim to finish around $11; above $16, another dollar is justified only if expected correctness improves by more than 0.2.	TR: mean $5.72, 77/78 within B (ZS 33/78); 22.6 parent steps (ZS 38.2); 4.4 wait or status-poll steps (ZS 16.1; heuristic).

E META-VERIFIER INFERENCE PROMPTS

Sol and Claude Opus 4.8 use the same Meta-Verifier prompt. We show it in parameterized form in Listing 1. The runtime fills the parent identity, budget, agent pool, concurrency, and mode fields before launching the parent agent. For readability, `[TASK]` and `[TRAJECTORIES]` denote the mounted inputs, which appear in the actual prompt as `/verification/instruction.md` and `/verification/rollouts/rollout_000..015/trajectory.json`, respectively. The parent-specific substitutions are shown below. The listings below use minor wording edits to express permissions explicitly.

Parent	`[META-VERIFIER]`	`[RUNTIME / MODEL]`
Sol	`codex_sol`	`codex / gpt-5.6-sol`
Claude Opus 4.8	`claude_opus_4_8`	`claude-opus-4-8[1m]`

The remaining fields specify the per-task dollar budget, the available verifier agents and their input/output prices, the maximum number of concurrent delegated calls, and the mode-specific instruction. No learned harness resources are mounted for these inference-only experiments. Both parents can delegate to Claude Code and Codex agents. The Sol prompt lists Opus 5, Sonnet 5, Haiku 4.5, Sol, and Luna; the Opus 4.8 prompt additionally lists Opus 4.8. Each rendered rate card also states the agents' context limits and input/output prices.

Pools and harness restrictions across experiments. The pools above describe the initial inference configurations. The archived harness-optimization experiments also include GPT-5.6 Terra in their six-agent pool (Section C). Optimized harnesses can further restrict the available verification actions. The selected Science harness H_{3c1} delegates only to Luna, using up to three parallel candidate audits and at most one conditional additional call (Section D.2). Thus the model list in Section 4.1 summarizes available resources across configurations; each run uses its own configured pool and harness restrictions.

E.1 SHARED PROMPT

Listing 1: Parameterized prompt shared by Sol and Claude Opus 4.8 as Meta-Verifiers. Bracketed fields are filled by the runtime.

```
# Meta-Verifier

Assess the 16 candidate trajectories as needed and select the rollout
    most
likely to have succeeded at the original task.

Construct a verification policy from the actual task, candidate
    contents and
likely failure modes, observable environment evidence, available
    verifier
agents and prices, uncertainty, and remaining budget. Gather direct
    evidence
when possible, delegate precise unresolved questions, aggregate
    conflicting
evidence explicitly, and stop when further verification has lower
    expected
value than its cost.

## Basic verification knowledge

The following are optional building blocks, not a required algorithm.
    You are free to
use, combine, modify, or discard them, and to invent a different
verification procedure when it is better suited to the task and budget.
Decide the verification policy from the actual input: the task
    requirements,
the contents and failure modes of the candidate trajectories, evidence
    that can
```

````
be obtained from the environment, available agents and prices,
    uncertainty, and
remaining budget.

When requesting numerical judgments, a useful task-agnostic scale is:

- 1-4: clearly incorrect or unusable
- 5-8: major errors, despite some useful progress
- 9-10: likely incorrect or missing important evidence
- 11-12: likely correct, but uncertain or weakly verified
- 13-16: substantially correct with minor gaps
- 17-20: very strong evidence of complete correctness

The boundary between likely failure and likely success is 10 versus 11.

### Pointwise LLM-as-a-Judge

Judge each candidate independently against the original task, then
    select the
candidate with the highest score. Calls can be parallelized.

```text
for i in candidates:
    score[i] = call_agent(task, [i], "Score this trajectory from 1 to
    20 for task success")
return argmax(score)
```

### Pairwise verification and tournament ranking

Compare two candidates directly. Ask for independent 1-20 scores for A
    and B;
do not force a winner when they are equally strong. A tournament can
    allocate
comparisons across candidates, avoid repeated pairs, update rankings
    from wins
and score margins, and spend additional rounds on candidates that
    remain
competitive or uncertain. Early comparisons should give every candidate
     useful
coverage. Later comparisons can focus on similarly ranked candidates,
    because
those matchups are most useful for separating plausible winners.

```text
initialize comparison counts, opponents, wins, ties, and score margins
while pairwise budget remains:
    if some candidates have little evidence:
        choose an untested pair that balances comparison counts
    else:
        choose an untested pair with similar current ranks,
        prioritizing candidates that are competitive or uncertain
    score_A, score_B = call_agent(task, [A, B], "Compare A and B; score
     each 1-20")
    record a win, loss, or tie from the two scores
    update both candidates' cumulative score margins
    rerank candidates using win rate together with score-margin
    evidence
    save the current best candidate at desired budget checkpoints
return the highest-ranked candidate at the final checkpoint
```

A simple fixed budget is up to `3N` pairwise calls for `N` candidates,
    with
````

````
usable rankings after `1N`, `2N`, and `3N` calls. Avoid comparing the same pair
twice unless repeated evidence is an intentional use of the remaining budget.

### Rubric-guided judging

First generate a small set of task-specific, equal-weight correctness criteria
from the original task. Then apply each criterion independently through either
pointwise or pairwise judging and average the criterion scores.

```text
criteria = call_agent(task, [], "Generate 3 distinct equal-weight correctness criteria")
for criterion in criteria:
    evaluate candidates pointwise or pairwise using only this criterion
aggregate the equal-weight criterion scores and select the best candidate
```

### Other useful verification methods that can be chosen based on the input

- Derive tests or adversarial cases from the task, execute them, and compare
  observed behavior with the required behavior.
- Check whether required artifacts exist, parse, compile, or satisfy cheap
  structural invariants before spending compute on deeper checks.
- Ask one agent to critique a candidate and another agent to judge it using the
  critique as evidence.
- Repeat independent judgments and aggregate scores or votes to reduce
  stochastic error.
- Begin with cheap agents or checks and define the evidence thresholds that
  trigger stopping or escalation to stronger agents.
- Combine independent reasoning, environment evidence, pointwise judgments,
  pairwise comparisons, rubrics, tests, and critiques in one policy.

Python 3 is available as `python3`. You are free to write Python or shell programs to
schedule calls, run independent calls in parallel, store intermediate
evidence, aggregate scores, and track spending. All verifier calls must use
`call_verifier.sh`. For example, Python can submit and wait for a call as
follows:

```python
import json
import subprocess
from pathlib import Path

request = {
    "action": "call",
    "payload": {
        "call_id": "judge-0-1",
        "agent_id": "claude_sonnet_5",
        "objective": "Compare the two trajectories for correctness and
    score each from 1 to 20.",
        "candidate_indices": [0, 1],
````
````

```
        "depends_on": [],
        "attachments": [],
    },
}
path = Path("/tmp/judge-0-1.json")
path.write_text(json.dumps(request))
submitted = json.loads(subprocess.check_output(
    ["/verification/call_verifier.sh", "submit", str(path)], text=True
))
result = json.loads(subprocess.check_output(
    ["/verification/call_verifier.sh", "wait", submitted["request_id"],
     "60"],
    text=True,
))
while result.get("pending"):
    result = json.loads(subprocess.check_output(
        ["/verification/call_verifier.sh", "wait", submitted["
    request_id"], "60"],
        text=True,
    ))
```

```
## Immutable execution contract

- Original task: [TASK]
- Candidate trajectories: [TRAJECTORIES]
- Meta-Verifier: [META-VERIFIER]
- Runtime/model: [RUNTIME / MODEL]
- Per-task verification budget B: [BUDGET], including
  your own token cost and every verifier-agent call you make.
- Budget cost uses normalized I/O accounting: every input-category
  token is charged at the visible input price and every output token at
     the
  visible output price. Provider cache billing is tracked separately
    outside
  the Meta-Verifier and does not change B.
- Your own model usage is included in `B`; use the live budget response
     as the
  authoritative remaining amount.
- Ground-truth labels, hidden evaluator tests, and reference solutions
    are
  unavailable and must not be sought outside `/verification`.
- You must select one of the 16 supplied rollouts. You are permitted to
     use independent
  reasoning or execution as verification evidence, but you must not
    submit a
  newly constructed solution.

[MODE-SPECIFIC INSTRUCTIONS]

You do not need to register or save a plan. The runtime automatically
    records
your complete trajectory, delegated calls, results, and spending. Use
`call_verifier.sh help` for the call schema. No particular verification
strategy is prescribed by the immutable contract.

### Learned harness resources

No additional learned harness resources are mounted.

## Tools and environment

- Read, search, create, and edit files in the current task workspace.
- Run shell commands and installed programs, including Python 3 and
    repository build and test commands.
```

````
- Internet access is unavailable.
- Call headless verifier agents with `/verification/call_verifier.sh` as
  described below.

## Verifier agents

You are free to call and use any of the verifier agents below as part of your
verification strategy. These are agent sessions, not bare LLM requests.
Choose an agent with `agent_id`, decide which candidate trajectories it should
receive, and instruct it through the `objective` field.

The native context column is the maximum context available to that agent.
Account for it when deciding how many and which trajectories to send.

Prices are frozen USD per 1M normalized input/output tokens;
reasoning tokens are included in output. All agents use the same API below.
[AVAILABLE VERIFIER AGENTS AND PRICES]

## Headless verifier-agent API

Each request launches the selected verifier agent non-interactively. Choose
the agent with `agent_id`, decide which candidate trajectories to provide with
`candidate_indices`, and specify what the agent should do in `objective`.

```bash
/verification/call_verifier.sh submit /tmp/request.json
# -> {"ok":true,"request_id":"<request_id>"}
/verification/call_verifier.sh wait <request_id> [seconds]
# repeat `wait` if the response contains "pending":true
```

Submit a headless verifier call as:

```json
{
  "action": "call",
  "payload": {
    "call_id": "unique-id",
    "agent_id": "agent_id from the table",
    "objective": "your verifier prompt",
    "candidate_indices": [0, 1],
    "depends_on": [],
    "attachments": []
  }
}
```

A completed `wait` returns the verifier's output in `result`.

For example, use Bash to ask Sonnet to compare trajectories 0 and 1:

```bash
cat > /tmp/compare-0-1.json <<'JSON'
{
  "action": "call",
  "payload": {
    "call_id": "compare-0-1-sonnet",
````
````

```
    "agent_id": "claude_sonnet_5",
    "objective": "Compare the supplied candidate trajectories against
    the original task. Determine which is more likely to have succeeded
     and explain the decisive evidence.",
    "candidate_indices": [0, 1],
    "depends_on": [],
    "attachments": []
  }
}
JSON

submit_result=$(/verification/call_verifier.sh submit /tmp/compare-0-1.
    json)
request_id=$(printf '%s' "$submit_result" | sed -n 's/.*"request_id"[[:
    space:]]*:[[:space:]]*"\([^"]*\)".*/\1/p')
[ -n "$request_id" ] || { printf '%s\n' "$submit_result" >&2; exit 1; }

while true; do
  result=$(/verification/call_verifier.sh wait "$request_id" 60)
  if ! printf '%s' "$result" | grep -Eq '"pending"[[:space:]]*:[[:space
    :]]*true'; then
    printf '%s\n' "$result"
    break
  fi
done
```

At most [MAXIMUM CONCURRENT VERIFIER CALLS] verifier-agent calls are permitted to be active at once; additional requests wait in the runtime queue. To run verifier agents concurrently, submit multiple requests with unique `call_id` values without waiting for completion, then call `wait` on each returned `request_id`.

Check live budget usage at any time:

```bash
/verification/call_verifier.sh budget
# -> budget_usd, spent_usd, remaining_budget_usd, and current
    concurrency
```

All verifier-agent calls must use `call_verifier.sh`; direct provider CLI/API calls are prohibited. Manage your reasoning and verifier calls so that their combined cost stays within the per-task verification budget.

Scored result

Write the selected rollout to `/verification/decision.json`:

```json
{
  "selected_rollout": 0
}
```

`selected_rollout` is the only required scored field and must be an integer from 0 through 15. Optional diagnostic fields are permitted, but the primary Best-of-N scorer uses only `selected_rollout`.

E.2 MODE-SPECIFIC INSTRUCTIONS

For the *Adaptive* condition, the runtime replaces `[MODE-SPECIFIC INSTRUCTIONS]` with:

```
Begin with an initial verification strategy covering direct checks,
verifier-agent calls, conditional branches, stopping rules, and the
    decision
rule. You do not need to write or register it. During execution, you
    are free to
intervene arbitrarily after observing intermediate evidence. You are
    free to revise
the strategy, change model allocation, launch additional or different
verifier calls, alter the order of checks, stop unproductive work, stop
     early
when evidence is sufficient, or escalate when results disagree or
    remain
ambiguous. Base these interventions on completed evidence and the
    remaining
budget. Your complete execution trace is preserved for later analysis.
```

For the *Plan-and-Execute* condition, it replaces the same field with:

```
First inspect the task and candidate trajectories and form a complete
verification plan covering direct reasoning/checks, Python or shell
    checks,
verifier-agent calls, conditional branches, stopping rules, and the
    final
decision rule. Settle this plan internally before making verifier-agent
     calls
or beginning environment-based verification; you do not need to write
    or
register it. Once execution begins, follow the complete plan without
replanning or changing its verification strategy, model allocation,
    call
sequence, stopping rule, or decision procedure in response to
    intermediate
results. Your complete execution trace is preserved for later analysis.
```

F HARNESS OPTIMIZATION DETAILS

F.1 SEARCH AND SELECTION

Sections 3.3 and 4.1 define the strict and relaxed training objectives for objective-specific runs. The archived initial $B = \$4$ run instructed the optimizer to improve selection accuracy, budget adherence, and realized cost jointly, without an explicit scalar utility. The validation-only results below retain the original accuracy-first selection rule.

Both budget conditions use the same six-agent pool, fixed model weights, execution protocol, and prices. The optimizer can revise prompts, skills, tools, memory, and verification strategies without a prescribed verification mechanism. It receives training traces, labels, and costs; reusable harnesses must exclude instance-specific answers and labels. Five disjoint 12-task training batches cover 60 tasks, with sibling harnesses evaluated on identical task–budget pairs. After search, all nine harnesses are ranked by accuracy on the same 54 validation tasks. Ties favor budget adherence, protocol adherence, and then lower cost, with H_0 preferred on an exact tie. Validation and test feedback are withheld from the optimizer.

F.2 RESULTS AND COST ACCOUNTING

Table 8 compares the initial and selected harnesses. Each task–harness pair is evaluated once; results are validation-only and do not estimate held-out performance or run-to-run variance. Delegation failures limit comparisons across budget conditions.

Table 8: Initial and selected ($\star$) harnesses on 54 validation tasks. Spend uses normalized experiment USD per task; $\geq$ denotes a lower bound from incomplete telemetry. Budget adherence counts recorded within-budget decisions, with unknown decisions receiving no credit.

Budget	Harness	Val. acc. (%)	Mean spend USD/task	Mean C/B (%)	Recorded budget adherence (%)
\$4	H_0	55.6	≥ 15.521	≥ 388.0	11.1
\$4	$H_{3c1}{}^{\star}$	61.1	2.823	70.6	85.2
\$16	H_0	55.6	17.192	107.5	38.9
\$16	$H_{3c1}{}^{\star}$	59.3	7.541	47.1	96.3

Verification cost includes the Meta-Verifier and all delegated calls at frozen uncached input/output prices; outer-loop optimization is accounted for separately. At $B = \$4$, only 49/54 initial-harness tasks are fully metered, so recorded charges establish lower bounds on mean spend and budget use. At $B = \$16$, mean costs follow the frozen experiment accounting, retaining recorded zero-cost entries despite incomplete provider telemetry. All tasks remain in the accuracy and adherence denominators. Mean spend below budget does not imply per-task compliance: the selected \$4 harness has eight overruns, while the selected \$16 harness has two failed tasks with unknown budget decisions.

G RESPONSIBLE-USE STATEMENT

Meta can improve coding-agent reliability and reduce manual review, but incorrect judgments can propagate errors or biases. More capable autonomous verifiers could also support harmful software development. Tool execution and trace sharing can expose sensitive information, and repeated verification increases resource demands. We recommend sandboxed execution with restricted permissions, redaction of sensitive trace content, externally enforced compute budgets, and human review for consequential decisions. Deployment should include monitoring for failures and misuse; benchmark performance alone does not establish safety or reliability beyond the evaluated tasks.