META^2: PRECOMPUTING REUSABLE KNOWLEDGE FOR ONLINE HARNESS EVOLUTION IN LONG-HORIZON AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Harness optimization improves long-horizon agents by refining the prompts, tools, memory, and hooks around a frozen model. Offline methods can learn from completed trajectories, but deploy one shared harness that cannot react to evidence revealed during a new run. Online evolution can react within the run, but may have to reason about every intervention from scratch. We introduce META^2, which bridges these regimes by precomputing reusable knowledge about *how to evolve a harness*, rather than precomputing one universal harness. **Offline**, META^2 compares three rollout conditions: the task agent alone, the current Meta-Skill-guided system, and a training-only evolver with privileged verifier information. These contrasts provide evidence that an evolver should `act`, `hold`, or `avoid` an intervention, which is aggregated across tasks into a textual Meta-Skill $\mathcal{M}$. **Online**, $\mathcal{M}$ is frozen. At checkpoints within an uninterrupted held-out run, the evolver combines it with active-run evidence and chooses KEEP or a task-local EVOLVE edit; the task agent then continues from the same environment state. Neither held-out verifier feedback nor task-specific edits propagate across runs. META^2 achieves the highest point estimates among the evaluated methods on both Terminal-Bench 2.1 and DeepSWE 1.1 with GPT-5.6-Terra, reaching 81.00% and 44.44%, respectively. These scores are 6.33 and 5.33 percentage points above the strongest comparators. With DeepSeek-V4-Flash-0731, META^2 achieves 77.67% on Terminal-Bench 2.1, 3.00 percentage points above the strongest comparator. Cumulative ablations associate the largest additional gains with contrastive Meta-Skill construction, while checkpoint sensitivity exposes a responsiveness–cost tradeoff. These results support precomputing reusable evolution knowledge while keeping each realized harness edit local to the active task.

1 INTRODUCTION

Long-horizon agents do not act through a model alone. They act through a *harness*: prompts that frame decisions, a context policy that preserves information, tools and hooks that enable actions, and checks for completion. These mechanisms are often fixed before execution, yet their need emerges during it. A missing tool becomes apparent only when the agent tries to use it; an overlooked constraint may surface in an intermediate artifact; and a one-time warning may need to become a recurring check. A harness chosen in advance must anticipate such situations before the trajectory that exposes them exists.

Because different tasks expose different interface and control needs, hand-designed harnesses must either anticipate a wide range of situations or be specialized manually. Automatic harness optimization reduces this burden, but most methods still make their consequential choices outside the active run. Offline methods optimize prompts, memory, tools, workflows, or complete harness programs on training tasks and deploy one shared artifact (Lee et al., 2026; Lin et al., 2026; Zhang et al., 2026b). Online methods adapt the harness after the target task becomes available: some customize it before execution or revise it between attempts (Zhang et al., 2026a; Shao et al., 2026), while recent systems modify prompts, tools, skills, or memory during execution (Karten et al., 2026; Fu et al., 2026; Xiao et al., 2026; Xia et al., 2025). Online evolution can occur at several points relative to execution. *Pre-run* adaptation customizes the harness from the target instruction before execution,

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

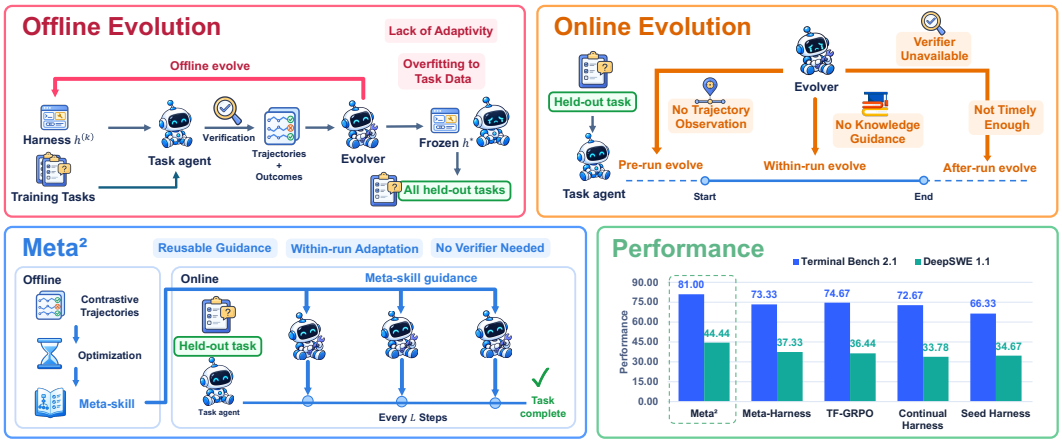


Figure 1: **Motivation and overview of META^2.** Offline evolution produces a fixed harness that cannot adapt to evidence revealed during a held-out run, while existing online-evolution regimes lack either timely trajectory evidence or reusable guidance. META^2 separates the two: it learns a reusable Meta-Skill offline from contrastive trajectories and uses it to guide task-local harness evolution at checkpoints online, without access to the held-out verifier. Across both evaluated benchmarks, this design yields the highest performance point estimate among the compared methods.

whereas *between-attempt* adaptation can use feedback from a completed attempt. Neither can apply an edit to the unfinished run in which the motivating evidence arises. *Within-run* evolution can, which raises our central design question: ***how can lessons learned from completed training runs guide selective edits during a new run, without revealing held-out verifier feedback or making one task's correction persist into unrelated tasks?***

This question is difficult because intervention is a sequential decision under partial information. At a checkpoint, an evolver can observe the active trajectory and permitted workspace state, but not the run's eventual verifier outcome. It must decide whether intervention is warranted, what harness component to change, and how to express the change. Acting can rescue a failing trajectory, but it can also interrupt productive work or introduce a new failure; sometimes the correct action is to do nothing. Offline supervision is also coarse: a terminal pass or failure evaluates a sequence of checkpoint decisions jointly rather than identifying which individual edit helped or harmed. Finally, reasoning about every intervention from scratch adds online token, latency, and monetary cost.

We introduce META^2, a precompute-and-reuse approach. Its design is to separate *reusable evolution knowledge* from the *task-local edit* produced with that knowledge. Offline, META^2 learns a textual Meta-Skill $\mathcal{M}$ from training trajectories and verifier outcomes. Online, every task begins from the same seed harness $h^{(0)}$, while $\mathcal{M}$ remains frozen. A task agent executes normally; at checkpoints, a separate evolver combines $\mathcal{M}$ with active-run evidence $\mathcal{I}^{(k)}$ and chooses KEEP or EVOLVE$(\delta^{(k)})$. Here, an edit $\delta^{(k)}$ is a bounded change to a notice, instruction, hook, or tool channel, and it affects only the remainder of that run. The task agent then continues with the same environment state and partial trajectory. Thus, costly-to-acquire experience transfers across tasks, whereas the resulting harness change remains conditioned on the task that needs it. Figure 1 illustrates this handoff, which we summarize as *rehearse offline, evolve online*.

The Meta-Skill is constructed from outcome contrasts rather than successful traces alone. For each training task, condition A uses the task agent alone, condition B uses the current Meta-Skill-guided evolver, and condition C uses a training-only evolver with privileged verifier information. A–B comparisons expose interventions that helped, harmed, or were unnecessary; B–C comparisons expose recoveries that the current evolver missed and privileged overreach. These contrasts provide directional evidence to `act`, `avoid`, or `hold`; they do not assign causal credit to a single checkpoint. Across training batches, REFLECT, AGGREGATE, and UPDATE convert recurring cross-task patterns into a compact Meta-Skill while removing instance-specific details. During online evolution, neither held-out verifier outcomes nor task-specific edits enter $\mathcal{M}$.

We evaluate META^2 on Terminal-Bench 2.1 and DeepSWE 1.1. With GPT-5.6-Terra, it obtains the highest performance point estimate among the evaluated methods on both benchmarks, reach-

ing 81.00% and 44.44%, respectively; with DeepSeek-V4-Flash-0731, it again obtains the highest Terminal-Bench point estimate among the evaluated methods, 77.67%. Cumulative ablations associate the first improvement with the online evolver and the largest additional improvement, especially on DeepSWE, with contrastive Meta-Skill construction. We also separate one-time offline construction from online deployment cost and study how checkpoint frequency trades responsiveness for cost. We report these numerical comparisons as point estimates. Our contributions are:

1. We formulate harness evolution by the time an edit is selected, the evidence then available, and the executions it affects. This taxonomy identifies META^2 as a hybrid: reusable knowledge is learned offline, while each edit is selected online within-run and remains task-local.
2. We develop a checkpoint-based interface that preserves the active environment state, together with an offline contrastive procedure that turns A/B/C rollout comparisons into reusable `act`, `hold`, and `avoid` guidance for KEEP/EVOLVE decisions. A targeted, provenance-preserving solicitation procedure probes for missing harm and recovery contrasts when natural rollouts are uniformly successful or unsuccessful.
3. We evaluate task performance, offline and online cost, component contributions, robustness across two backbones, and checkpoint-frequency sensitivity on two long-horizon benchmarks.

2 PROBLEM FORMULATION: HARNESS EVOLUTION

Task execution through a harness. (1) A task instance $\xi = (x, \mathcal{E}_x, V_x) \sim \mathcal{D}$ comprises a public instruction x, an interactive environment $\mathcal{E}_x$, and a verifier V_x whose criteria and outcome need not be observable to the agent or the evolver. (2) A task agent combines a model parameterized by θ with a harness h_t, comprising prompts, context policy, tools, skills, memory, and hooks that shape execution by, for example, preserving constraints, enabling actions, and validating artifacts. (3) At environment step t, $\mathcal{E}_x$ has state s_t, including the current workspace and task artifacts, and emits an observation o_t. Conditioned on the instruction, interaction history, and active harness, the agent selects an action according to $a_t \sim \pi_\theta(\cdot \mid x, \tau_{\le t}, h_t)$, where $\tau_{\le t} = (o_1, a_1, \ldots, a_{t-1}, o_t)$. The environment transitions via $P_x(\cdot \mid s_t, a_t)$, and the verifier returns $y = V_x(s_T)$ at termination.

Harness Evolution. Since the most useful harness depends on the task and on what happens during execution, we let an evolver g revise it from evidence. Starting from a seed harness $h^{(0)}$, decision $k \in \{1, \ldots, K\}$ proposes and applies an edit:

$$\delta^{(k)} \sim g\Big(\cdot \mid h^{(k-1)}, \mathcal{I}^{(k)}, \mathcal{M}\Big), \qquad h^{(k)} = \mathrm{Apply}\Big(h^{(k-1)}, \delta^{(k)}\Big), \tag{1}$$

where $\mathcal{I}^{(k)}$ is the evidence available to this decision, $\mathcal{M}$ is reusable knowledge about when and how edits help or harm (possibly empty), $h^{(k)}$ is the harness after decision k, and a null edit sets $h^{(k)} = h^{(k-1)}$. We evaluate harness evolution by $V_x(s_T)$ on held-out tasks. Appendix Table 7 compares the offline and online instantiations of equation 1, which differ in available evidence and edit lifetime.

Offline evolution applies equation 1 on a training set $\mathcal{S}_{\mathrm{tr}}$ before any target task is observed. Its evidence consists of completed training trajectories and their verifier outcomes, and it outputs a single harness $h^\star$ shared by all target tasks. This amortizes experience across tasks, but $h^\star$ is fixed once execution begins: it cannot react to evidence revealed during a target run, and its persistent edits may overgeneralize beyond the failures that motivated them.

Online evolution applies equation 1 during a target run, at checkpoints $0 < t_1 < \cdots < t_K \le T$. Its evidence is limited to what the run has revealed: $\mathcal{I}^{(k)}$ may include x, $\tau_{\le t_k}$, and permitted workspace observations, but never access to the task-specific verifier V_x or its terminal outcome $y = V_x(s_T)$. The seed harness $h^{(0)}$ is active before t_1; after decision k, $h^{(k)}$ remains active until the next checkpoint or termination. Each edit is discarded when the run ends. Online evolution can therefore help the active attempt, but without reusable guidance it must reason about each intervention from scratch. **The two regimes are complementary**: offline evolution has verified experience but no view of the target run, whereas within-run online evolution observes the target run but, when $\mathcal{M}$ is empty, lacks reusable verified guidance. This raises the question of *whether verified offline experience can guide online edits without making any task-specific change persistent*.

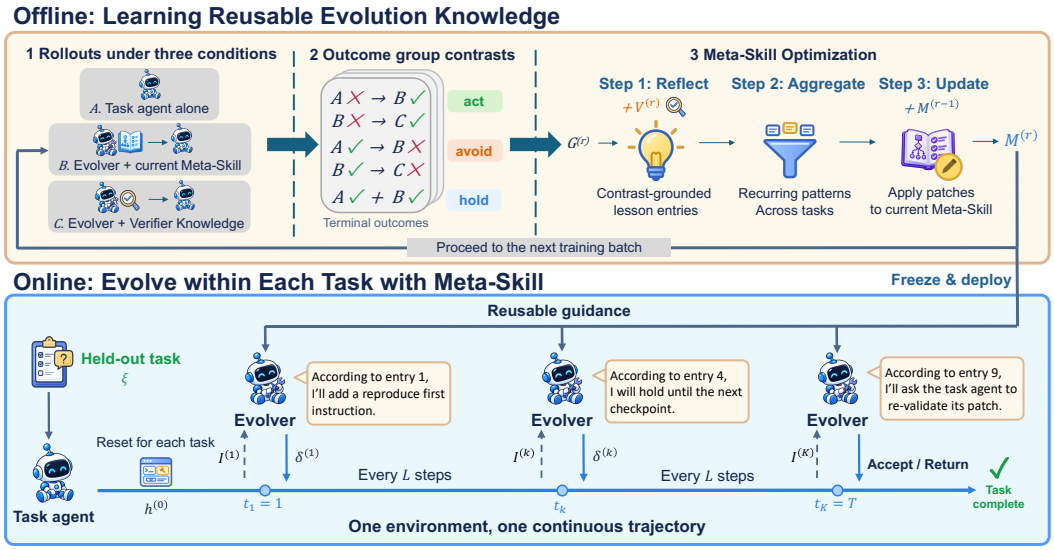


Figure 2: **Implementation flow for offline Meta-Skill learning and online harness evolution.** Offline, LLM agents collect A/B/C rollouts and deterministic outcome pairing creates `act`, `avoid`, and `hold` contrasts. LLM calls then write contrast-grounded lessons, merge recurring lessons across tasks, and update the Meta-Skill. Online, the frozen Meta-Skill guides an evolver LLM at deterministic checkpoints while the same task-agent LLM continues in one environment. The verifier is unavailable online.

3 META^2: PRECOMPUTING REUSABLE KNOWLEDGE FOR ONLINE EVOLUTION

META^2 answers this question by moving the offline stage one level up. Instead of optimizing a shared harness, it learns reusable evolution knowledge $\mathcal{M}$, the *Meta-Skill*, from training trajectories and verifier feedback, freezes it, and supplies it to online evolution. Experience thus transfers across tasks, while every edit remains local to its run. The method has three components (Figure 2): a checkpoint-based interface for within-run evolution, an offline procedure that learns $\mathcal{M}$, and a deployment protocol that reuses $\mathcal{M}$ without held-out verifier feedback.

3.1 ONLINE COMPONENT: CHECKPOINT-BASED HARNESS EVOLUTION

Challenge: intervene without interrupting. The evolver must decide whether and how to intervene before an active attempt ends, without resetting its environment, discarding its trajectory, or taking over its actions. META^2 therefore separates two roles instantiated from the same frozen language model. The **task agent** follows π_θ inside a minimal seed harness $h^{(0)}$, while the **evolver** g observes the run at checkpoints and may edit the harness. The task agent continues uninterrupted with the same environment state s_t and partial trajectory $\tau_{\le t}$.

Checkpoint information. Suppressing task and attempt indices j, m, let $\omega^{(k)}$ denote the permitted read-only workspace observations obtained at checkpoint k; it may be empty. The evolver also receives the newly accumulated trajectory $\Delta\tau^{(k)}$ while retaining its preceding checkpoint conversation. Its information therefore accumulates as

$$\mathcal{I}^{(k)} = \begin{cases} \big(x, \Delta\tau^{(1)}, \omega^{(1)}\big), & k = 1, \\ \mathcal{I}^{(k-1)} \oplus \big(\Delta\tau^{(k)}, \omega^{(k)}, \sigma^{(k-1)}\big), & 1 < k \le K. \end{cases} \tag{2}$$

where $\oplus$ denotes chronological append and $\sigma^{(k-1)}$ records whether the preceding edit was loaded, retained, or rolled back. This recursive construction operationalizes the regime-level $\mathcal{I}^{(k)}_{j,m}$ in Appendix Table 7: the trajectory deltas reconstruct $\tau_{j,m,\le t_k}$, while $\omega^{(k)}$ and $\sigma^{(k-1)}$ supply the workspace and edit-status evidence.

Decision and continuation. At checkpoint k, the evolver makes a binary top-level decision:

$$u^{(k)} \in \left\{ \textsc{Keep}, \textsc{Evolve}\left(\delta^{(k)}\right) \right\}. \tag{3}$$

KEEP leaves $h^{(k)}$ unchanged and continues execution; EVOLVE applies $\delta^{(k)}$ through a bounded notice, prompt, hook, or tool channel. At the terminal checkpoint, the evolver also receives mechanically collected workspace state. The action space remains unchanged: KEEP accepts the artifact, whereas EVOLVE applies a bounded correction and returns the task agent for more work. This realizes Equation 1 as the same-run loop $\left(h^{(k)}, \mathcal{I}^{(k)}, \mathcal{M}\right) \to u^{(k)} \to h^{(k+1)}$. Appendix B.2 specifies the checkpoint schedule, observation permissions, terminal state, and edit semantics.

3.2 OFFLINE COMPONENT: PRECOMPUTING REUSABLE EVOLUTION KNOWLEDGE

Challenge: learn checkpoint guidance from terminal outcomes. At deployment, the evolver must choose whether and how to edit from the active trajectory alone. Offline, the verifier supplies an outcome only after the entire run, evaluating a sequence of checkpoint decisions jointly. A passing or failing rollout therefore does not identify which intervention helped, harmed, or was unnecessary. Even when a useful intervention is found, its lesson must be separated from task-specific details before it can transfer.

Optimize the knowledge, not a single harness. We hold the language model, seed harness, checkpoint interface, and edit channels fixed, and optimize only the evolver's reusable knowledge $\mathcal{M}$. This Meta-Skill links observable execution conditions to three guidance types: `act` supports an EVOLVE decision, `hold` supports KEEP, and `avoid` rules out a specific intervention even when another edit may remain appropriate. At deployment, the evolver combines $\mathcal{M}$ with the evidence available at checkpoint k, $\mathcal{I}^{(k)}$, to produce a task-specific harness edit $\delta^{(k)}$. Offline optimization therefore transfers decision knowledge, not a universal harness or a predetermined edit. Appendix B.1 gives the full offline construction algorithm, and Appendix H specifies the entry fields.

A/B/C rollouts make terminal feedback informative. For each training task, we execute and compare three rollout conditions with the same language model and seed harness: *A*, the task agent alone; *B*, the current Meta-Skill-guided system; and *C*, a training-only evolver with privileged verifier information. A–B contrasts measure the current evolver's effect, whereas B–C contrasts expose missed or excessive interventions.

Five contrast directions. B-pass/A-fail and C-pass/B-fail provide `act` evidence for effective and missed informed interventions. B-fail/A-pass and C-fail/B-pass provide `avoid` evidence for harm and privileged overreach. A-pass/B-pass provides `hold` evidence for unnecessary intervention. These comparisons give each candidate lesson a direction, not causal credit for one checkpoint; reflection still examines the paired trajectories, evolution histories, and outcomes.

Targeted solicitation of missing contrasts. Natural rollouts provide no directional contrast when every condition passes or every condition fails. A major technical contribution of META^2 is a provenance-preserving solicitation procedure that creates additional opportunities to observe informative discordance without changing held-out evaluation. After a fixed natural-rollout budget, we use isolated training replicas for tasks with no discordant outcomes. An induced-disruption prompt probes harm on all-pass tasks, while a verifier-assisted evolver probes missed recoveries on all-fail tasks. Solicitation does not presume success: every attempt is tagged and retained, including unchanged and unsuccessful outcomes. Appendix B.1 specifies the solicitation logic, and Appendix H specifies the record fields and leakage-audit requirements.

From task-level contrasts to reusable rules. We process training tasks in batches. For batch r, REFLECT inspects each retained contrastive group and writes candidate entries; AGGREGATE removes instance-specific details by fusing patterns that recur across tasks; and UPDATE adds, merges, refines, or deletes entries relative to $\mathcal{M}^{(r-1)}$. Let $\mathcal{G}^{(r)}$ denote the retained contrastive groups and $\mathcal{V}^{(r)}$ their privileged training-time verifier information. The update is

$$\mathcal{M}^{(r)} = f_{\text{upd}}\left(\mathcal{M}^{(r-1)}, f_{\text{agg}}\left(f_{\text{ref}}\left(\mathcal{G}^{(r)}, \mathcal{V}^{(r)}\right)\right)\right). \tag{4}$$

The contrast supplies the direction of the lesson; cross-task aggregation supplies its claim to reuse. Repeating the update across batches precomputes general evolution knowledge once, while leaving each deployed harness edit conditioned on the active run.

3.3 Deployment Component: Frozen Knowledge and Task-Local Edits

Challenge: reuse without contamination. Deployment must reuse offline experience without exposing held-out verifier feedback or allowing one task's correction to alter later tasks.

Freeze globally, adapt locally. At deployment, the learned Meta-Skill $\mathcal{M}$ is frozen. Every task starts from the same seed harness $h^{(0)}$; at checkpoint k, the evolver combines $\mathcal{M}$ with $\mathcal{I}^{(k)}$ and chooses KEEP or EVOLVE$(\delta^{(k)})$ as described in Section 3.1. The resulting edit affects only the active run. Neither held-out verifier feedback nor an LLM-as-a-judge signal is used to update $\mathcal{M}$ after the task. Thus, general evolution knowledge is learned once and amortized across tasks, while task-specific adaptation cannot propagate to unrelated runs.

4 Experiments

We organize the evaluation around four questions. **RQ1** asks whether META^2 improves held-out task performance. **RQ2** examines whether the offline cost of precomputing reusable evolution knowledge is justified when reused at deployment. **RQ3** studies the tradeoff between checkpoint responsiveness and online cost. **RQ4** identifies which method components account for the gains.

4.1 Experimental setup

Benchmarks and metrics. We evaluate META^2 on Terminal-Bench 2.1 (Merrill et al., 2026) and DeepSWE 1.1 (Huang et al., 2026). We randomly split each benchmark into training and test sets at a 1:2 ratio. We report mean@5 over five independent runs on every held-out Terminal-Bench task and mean@3 over three independent runs on every held-out DeepSWE task. Appendix E reports 95% paired, task-level percentile bootstrap interval half-widths for differences from the NexAU seed, together with unpaired absolute-score intervals for the seed itself. Appendix C provides dataset splits, repetition counts, and additional evaluation and reporting details.

Baselines. We compare META^2 against three groups of baselines: **(1) human-designed harnesses**: NexAU (nex-agi, 2025), Codex CLI (OpenAI, 2025), mini-swe-agent (Yang et al., 2024b), Pi agent (Earendil Inc., 2025), and DeepSeek Harness (DeepSeek-AI, 2026a); **(2) self-evolution methods**: ACE (Zhang et al., 2026d), Training-free GRPO (Cai et al., 2025), and ReasoningBank (Ouyang et al., 2026); and **(3) harness-evolution methods**: AHE (Lin et al., 2026), Meta-Harness (Lee et al., 2026), and Continual Harness (Karten et al., 2026). All self-evolution and harness-evolution methods start from the same NexAU seed harness, denoted $h^{(0)}$ for META^2.

Implementation details. Self-evolution baselines build one fixed training-split experience library. AHE uses best-of-two proposals for five iterations. Meta-Harness uses one proposal for ten iterations; both deploy their best training harness unchanged. Continual Harness resets each test run to its training-evolved harness, whereas META^2 resets to $h^{(0)}$ with a frozen Meta-Skill. Both may adapt only within the active run. No test-time method observes the held-out verifier outcome. Appendix C gives the full settings.

LLM backbones. Our primary experiments use GPT-5.6-Terra (OpenAI, 2026) with medium reasoning effort. We additionally evaluate Terminal-Bench 2.1 with DeepSeek-V4-Flash-0731 (DeepSeek-AI, 2026b) at high reasoning effort. Within each backbone setting, all agent roles across all compared systems use the same model. We route all calls through one LiteLLM proxy and compute cost from the per-call charges in its response headers. Appendix C.3 provides additional protocol and reporting details.

4.2 RQ1: Does META^2 Improve Task Performance?

Primary comparison. META^2 achieves the highest point estimate on both benchmarks with GPT-5.6-Terra: Table 1 reports 81.00% on Terminal-Bench 2.1 and 44.44% on DeepSWE 1.1. Compared with Meta-Harness, the strongest harness-evolution baseline, META^2 obtains larger point-estimate gains over the shared NexAU seed harness: 14.67 versus 7.00 points on Terminal-Bench and 9.77

versus 2.66 points on DeepSWE. All evolution-based methods improve over the seed on Terminal-Bench. On DeepSWE, however, ReasoningBank, ACE, and Continual Harness fall below the seed, while AHE only matches it. In contrast, META^2 retains a 9.77-point improvement, suggesting that the favorable point-estimate comparison is not specific to Terminal-Bench.

Robustness across backbones. Table 2 repeats the Terminal-Bench evaluation with DeepSeek-V4-Flash-0731 as the backbone. META^2 again achieves the highest point estimate, reaching 77.67% and outperforming the shared NexAU seed and Continual Harness by 7.67 and 3.00 points, respectively. The three self-evolution baselines fall below the seed; AHE and Meta-Harness improve by 1.33 and 1.00 points, and Continual Harness improves by 4.67 points. Thus, META^2 retains the highest point estimate under a second backbone. Because the offline knowledge is constructed within each backbone setting, this experiment measures robustness to backbone choice rather than zero-shot transfer of one frozen Meta-Skill between models.

Table 1: **Main performance and cost comparison.** Results with GPT-5.6-Terra (medium reasoning) on Terminal-Bench 2.1 (mean@5) and DeepSWE 1.1 (mean@3). Parentheses report the point change from the NexAU seed; costs are aggregate US dollars over each evaluation.

Method	Terminal-Bench 2.1				DeepSWE 1.1			
	Perf.	Cost ($)			Perf.	Cost ($)		
		Offline	Online	Full		Offline	Online	Full
Human-Designed Harnesses								
NexAU Seed	66.33	–	34.09	34.09	34.67	–	62.67	62.67
Codex CLI	68.33	–	56.59	56.59	31.11	–	136.53	136.53
mini-swe-agent	67.33	–	41.08	41.08	29.33	–	68.87	68.87
Pi agent	69.00	–	37.98	37.98	39.11	–	82.50	82.50
DeepSeek Harness	68.00	–	31.90	31.90	34.22	–	41.37	41.37
Self-Evolution Methods								
ReasoningBank	68.00 (+1.67)	16.04	55.60	71.64	33.33 (-1.34)	55.87	85.64	141.51
ACE	71.33 (+5.00)	81.55	64.09	145.64	32.00 (-2.67)	238.10	82.29	320.39
TF-GRPO	74.67 (+8.34)	92.55	42.62	135.17	36.44 (+1.77)	322.50	86.16	408.66
Harness-Evolution Methods								
AHE	70.67 (+4.34)	209.44	38.91	248.35	34.67 (0.00)	793.48	69.41	862.89
Meta-Harness	73.33 (+7.00)	160.51	38.34	198.85	37.33 (+2.66)	781.56	89.19	870.75
Continual Harness	72.67 (+6.34)	28.75	83.48	112.23	33.78 (-0.89)	46.11	155.61	201.72
META^2	**81.00** (+14.67)	110.19	64.72	174.91	**44.44** (+9.77)	238.52	129.34	367.86

Table 2: **Robustness across backbones.** Terminal-Bench 2.1 test-set results with DeepSeek-V4-Flash-0731 (high reasoning), reported as mean@5. Parentheses report the point change from the NexAU seed.

Method	Perf.	Cost ($)		
		Offline	Online	Full
NexAU Seed	70.00	–	16.83	16.83
Self-Evolution Methods				
ReasoningBank	69.33 (-0.67)	8.08	21.21	29.29
ACE	65.00 (-5.00)	58.30	21.56	79.86
TF-GRPO	66.33 (-3.67)	37.15	19.86	57.01
Harness-Evolution Methods				
AHE	71.33 (+1.33)	123.51	20.28	143.79
Meta-Harness	71.00 (+1.00)	99.55	19.49	119.04
Continual Harness	74.67 (+4.67)	2.84	35.98	38.82
META^2	**77.67** (+7.67)	81.90	33.51	115.41

4.3 RQ2: Is Precomputation Cost-Effective When Reused?

Cost comparison. Across both benchmarks, META^2 combines the highest performance point estimate with a non-dominated full cost. Table 1 separates the one-time offline construction cost from the online cost of the held-out runs. META^2's offline costs are \$110.19 on Terminal-Bench and \$238.52 on DeepSWE, below AHE (\$209.44 and \$793.48) and Meta-Harness (\$160.51 and \$781.56). Although its two-agent deployment costs more than a single-agent harness, its online cost remains below that of Continual Harness on both benchmarks (\$64.72 vs. \$83.48 and \$129.34 vs. \$155.61). Consequently, META^2 lies on the point-estimate performance–full-cost Pareto frontier in both evaluations.

Reusable precomputation. The full-cost results charge the entire Meta-Skill construction cost to the evaluated workload, even though the resulting knowledge can be reused without reconstruction across subsequent tasks and runs. After this one-time construction step, additional deployments incur only the online cost. Table 2 illustrates this distinction: with DeepSeek, META^2 has a slightly lower online cost than Continual Harness (\$33.51 vs. \$35.98), despite having a higher full cost when Meta-Skill construction is included. Thus, the initial precomputation enables repeated deployment at a competitive marginal cost.

4.4 RQ3: How Does Checkpoint Frequency Trade Responsiveness for Cost?

Checkpoint-interval sensitivity. Figure 3 varies the fixed checkpoint interval L on Terminal-Bench 2.1. A moderate interval of $L = 5$ achieves the highest performance (81.00%) and the best observed performance–cost balance, at an online cost of \$64.72. More frequent checkpoints increase cost without improving performance: $L = 3$ and $L = 1$ cost \$74.22 and \$83.41 but score 80.33% and 79.00%. Longer intervals reduce cost modestly, reaching \$59.10 at $L = 11$, but all underperform $L = 5$. We therefore use $L = 5$ by default. Importantly, META^2 outperforms every baseline at every evaluated value of L, showing that its advantage is robust to the checkpoint schedule.

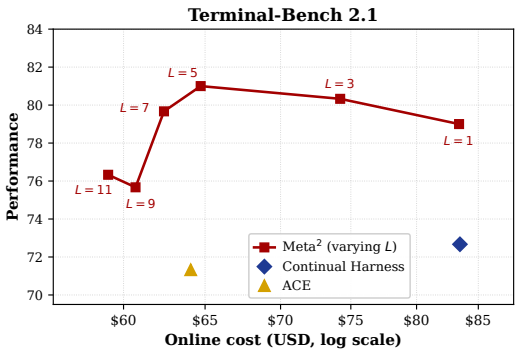


Figure 3: **Checkpoint-interval sensitivity.** Terminal-Bench 2.1 performance versus log-scaled online cost.

4.5 RQ4: Which Components Produce the Gains?

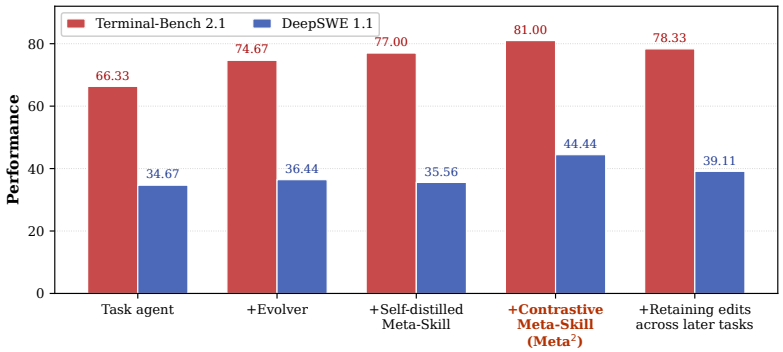


Figure 4: **Cumulative component ablation.** Each point adds one component to the preceding configuration.

Component contributions. Figure 4 compares five settings. The first four cumulatively add the online evolver, a self-distilled Meta-Skill, and the full contrastive Meta-Skill to the task agent. Their

Terminal-Bench/DeepSWE scores are 66.33/34.67, 74.67/36.44, 77.00/35.56, and 81.00/44.44, respectively. Thus, the online evolver provides the first improvement, while contrastive Meta-Skill learning produces the largest additional gain, particularly on DeepSWE.

Resetting between tasks. The fifth setting starts each new task from the harness produced by the preceding task, retaining all edits made during online evolution rather than resetting to $h^{(0)}$. This cross-task retention reduces performance relative to the full META^2 configuration. Online edits are derived from the trajectory of the active task and may therefore encode task-specific interventions that do not transfer to the next task. The result supports resetting the harness at the beginning of every task: reusable knowledge is carried across tasks through the frozen Meta-Skill, while task-conditioned harness edits remain local to the run that produced them.

5 Related Work

Offline evolution. Using the taxonomy in Section 2 and Appendix Table 7, we group prior work by edit selection time. Offline methods optimize the system around a frozen model on training tasks and deploy the result as a fixed artifact. Early work optimizes prompts and LM pipelines (Yang et al., 2024a; Khattab et al., 2023; Yuksekgonul et al., 2024; Agrawal et al., 2026), and later work searches over code-defined agent systems and workflows (Hu et al., 2025; Zhang et al., 2025). For long-horizon agents, recent methods evolve the harness itself: they rewrite the scaffold (Zhang et al., 2026c), propose harnesses from prior code, scores, and traces (Lee et al., 2026), tie edits to checkable predictions (Lin et al., 2026), or retain regression-tested edits (Zhang et al., 2026b). Analyses suggest that many of these gains compensate for recoverable execution defects of the underlying model (Yang et al., 2026). In all cases, the deployed harness $h^\star$ is shared across held-out runs and cannot react to evidence revealed during a run. META^2 also learns from completed training runs, but it freezes reusable evolution knowledge $\mathcal{M}$ rather than a harness, so the same $\mathcal{M}$ can support different edits in runs.

Online evolution: pre-run and after-run. Pre-run methods adapt the harness once the target task is known but before execution begins; for example, JIT-Agent synthesizes a task-specific harness for each incoming task (Zhang et al., 2026a). After-run methods adapt from completed attempts. Reflexion (Shinn et al., 2023) and Harness-R1 (Shao et al., 2026) use feedback or failure trajectories to improve later attempts on the same task, while Dynamic Cheatsheet, ACE, and TTHE update memory, guidance, or harness programs over a stream of tasks (Suzgun et al., 2026; Zhang et al., 2026d; Nie et al., 2026). In both settings, the edit either precedes the evidence that execution reveals or benefits only later attempts or tasks, and after-run selection often relies on completed-run feedback or proxy signals. META^2 instead edits the run in which the evidence appears, without observing that run's verifier outcome.

Online evolution: within-run. The closest setting adapts the harness during an uninterrupted run. Continual Harness refines prompts, skills, and memory inside a reset-free episode (Karten et al., 2026); PILOT uses a supervisor to steer an active worker (Xiao et al., 2026); and Live-SWE-agent modifies its own tools while solving a task (Xia et al., 2025). These systems show that within-run adaptation is feasible and useful, but they decide when and how to intervene without learned guidance. META^2 supplies this guidance. Offline, A/B/C outcome contrasts produce a textual Meta-Skill that records when to act, hold, or avoid an intervention. Online, a separate evolver applies this frozen knowledge to the active run without held-out verifier feedback. Appendix Table 6 summarizes the distinction: META^2 is a hybrid that evolves reusable knowledge offline and task-local edits within the run.

6 Conclusion

META^2 learns reusable knowledge for online harness evolution rather than one fixed harness. Offline outcome contrasts produce a frozen Meta-Skill, which an online evolver uses to keep or modify the harness within the active run. META^2 achieves 81.00% on Terminal-Bench 2.1, 44.44% on DeepSWE 1.1, and 77.67% under the DeepSeek backbone, with ablations highlighting the value of contrastive Meta-Skill learning. These results support a simple principle: precompute *how to adapt*, then adapt locally as the task unfolds.

AI USE STATEMENT

Generative AI tools assisted with software implementation and code editing, literature search and reference checking, interpretation of reported results, figure preparation, translation, and manuscript drafting and revision. The experiments also use language models as agents and editors, as described in Section 4. The authors take responsibility for the final text, claims, code and other research artifacts.

ETHICS STATEMENT

Online harness evolution can amplify both useful and harmful agent behavior. A supervisor that edits prompts, tools, or middleware may expand permissions, introduce insecure code, expose private trajectory data, or conceal the source of an action. Our proposed safeguards include read-only supervision by default, restricted and auditable edit scopes, validation before activation, versioned changes, and rollback. Held-out evaluation answers and hidden verifier details must be excluded from the reusable Meta-Skill to prevent benchmark leakage. Offline training feedback may support lesson construction, but deployment rules must rely on evidence observable during the task. Deliberate-disruption prompts are restricted to isolated training replicas and never alter the evaluator. Verifier-assisted oracle branches are labeled diagnostics; held-out oracle artifacts cannot enter ordinary training or deployment references. Deployments in consequential settings should additionally require human authorization for actions that change permissions or affect external systems.

REPRODUCIBILITY STATEMENT

Section 2 defines the components and objective, Section 3 specifies natural and solicited contrast construction, the offline aggregation and update workflow, and the binary online modification decision, and Section 4 reports the evaluation setup, performance and cost results, component ablations, and checkpoint-interval sensitivity. Appendix B specifies the offline and online algorithms. Appendix C provides dataset splits, model configurations, baseline settings, and implementation details. The final artifact release will include versioned Meta-Skills and their source overviews, harness versions, prompts, intervention logs, evaluation commands, and per-task verifier outputs.

REFERENCES

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.

Yuzheng Cai, Siqi Cai, Yuchen Shi, Zihan Xu, Lichao Chen, Yulei Qin, Xiaoyu Tan, Gang Li, Zongyi Li, Haojia Lin, et al. Training-free group relative policy optimization. *arXiv preprint arXiv:2510.08191*, 2025.

DeepSeek-AI. DeepSeek Harness: Everything is a plugin. https://github.com/deepseek-ai/deepseek-harness, 2026a.

DeepSeek-AI. DeepSeek-V4: Towards highly efficient million-token context intelligence, 2026b.

Earendil Inc. Pi: A minimal coding agent harness. https://github.com/earendil-works/pi, 2025. Accessed 2026-09-24.

Jiarun Fu, Lizhong Ding, Sida Chen, Honglei Xin, Chunhui Zhang, Pengqi Li, Qiuning Wei, Ye Yuan, and Guoren Wang. CHILL-Harness: Counterfactual harness learning for efficient reasoning in long-horizon agents. *arXiv preprint arXiv:2607.25825*, 2026.

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.

Seth Karten, Joel Zhang, Tersoo Upaa Jr, Ruirong Feng, Wenzhe Li, Chengshuai Shi, Chi Jin, and Kiran Vodrahalli. Continual harness: Online adaptation for self-improving foundation agents. *arXiv preprint arXiv:2605.09998*, 2026.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, et al. DSPy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2023.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-Harness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052*, 2026.

Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Zhiheng Xi, Xuanjing Huang, Hang Yan, Zhenhua Han, Tao Gui, et al. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses. *arXiv preprint arXiv:2604.25850*, 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.

nex-agi. NexAU: A general-purpose agent framework. https://github.com/nex-agi/NexAU, 2025. Version 0.4.1. Accessed 2026-09-24.

Jun Nie, Yonggang Zhang, Jun Song, Qianshu Cai, Dahai Yu, Yike Guo, Xinmei Tian, and Bo Han. TTHE: Test-time harness evolution. *arXiv preprint arXiv:2607.08124*, 2026.

OpenAI. Codex: A lightweight coding agent that runs in your terminal. https://github.com/openai/codex, 2025. GitHub repository. Accessed: 2026-09-25.

OpenAI. GPT-5.6. https://openai.com/index/gpt-5-6/, 7 2026. GPT-5.6 Terra variant.

Siru Ouyang, Jun Yan, I Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long Le, Samira Daruki, Xiangru Tang, et al. ReasoningBank: Scaling agent self-evolving with reasoning memory. In *International Conference on Learning Representations*, volume 2026, pp. 94327–94354, 2026.

Shuai Shao, Kangning Zhang, Qingyao Li, Shijian Wang, Hao Wang, Wenxiang Jiao, Yuan Lu, Yi Guo, Weiwen Liu, and Weinan Zhang. Harness-R1: Learning to edit executable runtime harnesses from agent failure trajectories. *arXiv preprint arXiv:2608.02276*, 2026.

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

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic cheatsheet: Test-time learning with adaptive memory. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7080–7106, 2026.

Chunqiu Steven Xia, Zhe Wang, Yan Yang, Yuxiang Wei, and Lingming Zhang. Live-swe-agent: Can software engineering agents self-evolve on the fly? *arXiv preprint arXiv:2511.13646*, 2025.

Yang Xiao, Yusong Sun, Haoyi Wu, Wenyang Hui, Wen Da, Zhaokai Luo, Mu Chuan, Yao Hu, Wenjie Li, and Chengyue Jiang. PILOT in the loop: Live self-improvement for long-horizon agents. *arXiv preprint arXiv:2608.26530*, 2026.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *International Conference on Learning Representations*, volume 2024, pp. 12028–12068, 2024a.

John Yang, Carlos 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*, 37:50528–50652, 2024b.

Siqi Yang, Qianlan Yang, Yu-Xiong Wang, Saurabh Pujar, and Martin Hirzel. One recipe, many harnesses: What self-evolution encodes across languages and models. *arXiv preprint arXiv:2608.10178*, 2026.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. TextGrad: Automatic "differentiation" via text. *arXiv preprint arXiv:2406.07496*, 2024.

Guibin Zhang, Leo Lu, Fangzhou Xie, Kang Zhu, Junhao Wang, Zhifei Xie, Zhaochen Yu, Zihang Liu, Zhongxiang Sun, Qiankun Li, et al. JIT-Agent: Scaling harness intelligence via just-in-time harness evolution. *arXiv preprint arXiv:2608.25593*, 2026a.

Hangfan Zhang, Shao Zhang, Kangcong Li, Chen Zhang, Yang Chen, Yiqun Zhang, Lei Bai, and Shuyue Hu. Self-harness: Harnesses that improve themselves. *arXiv preprint arXiv:2606.09498*, 2026b.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin gödel machine: open-ended evolution of self-improving agents. In *International Conference on Learning Representations*, volume 2026, pp. 104223–104294, 2026c.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, et al. AFlow: Automating agentic workflow generation. In *International Conference on Learning Representations*, volume 2025, pp. 34040–34077, 2025.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, et al. Agentic context engineering: Evolving contexts for self-improving language models. In *International Conference on Learning Representations*, volume 2026, pp. 86069–86100, 2026d.

A NOTATION

Table 3 consolidates the notation used in Sections 2–3.

Table 3: Core notation used throughout the paper.

Symbol	Meaning
$\xi = (x, \mathcal{E}_x, V_x)$	Task instance.
s_t, o_t, a_t	State, observation, and action at step t.
$\tau_{\leq t}$	Partial trajectory through step t.
π_θ	Frozen task-agent policy.
$h^{(0)}$	Seed harness.
$h^{(k)}$	Harness after decision k.
g	Evolver policy.
$\Delta\tau^{(k)}$	Trajectory since the previous checkpoint.
$\omega^{(k)}$	Read-only workspace evidence.
$\sigma^{(k)}$	Previous-edit status.
$\mathcal{I}^{(k)}$	Evolver information at checkpoint k.
$\delta^{(k)}$	Task-local harness edit.
$u^{(k)}$	KEEP/EVOLVE decision.
$\mathcal{M}^{(r)}$	Meta-Skill after offline batch r.
$\mathcal{R}_\xi^q$	Rollouts under condition q.
$\mathcal{G}^{(r)}$	Admitted contrast groups.
L	Checkpoint interval.

B ALGORITHMIC DETAILS

This appendix gives executable-level specifications for the online checkpoint loop and offline Meta-Skill construction. The descriptions define the protocol used by the reported system; prompt templates are collected separately in Appendix H.

B.1 OFFLINE META-SKILL CONSTRUCTION ALGORITHM

Algorithm 1 Offline construction of reusable evolution knowledge

Require: Training batches $\{\mathcal{B}_r\}_{r=1}^{R}$; seed harness $h^{(0)}$; fixed task agent and evolver; checkpoint interface; edit channels
Ensure: Frozen Meta-Skill $\mathcal{M}$

```
Initialize M^(0) ← ∅
for r = 1, ..., R do
    for all ξ ∈ B_r do
        Load fixed condition-A records R_ξ^A
        Collect condition-B records R_ξ^B using M^(r−1)
        Load fixed condition-C records R_ξ^C and run the leakage audit
        if natural rollouts contain no eligible discordant group then
            Run the tagged disruption or verifier-assisted solicitation permitted for ξ
        end if
```

Algorithm 1 (continued)

$\mathcal{G}_\xi \leftarrow$ BUILDCONTRASTS$(\mathcal{R}^{\mathrm{A}}_\xi, \mathcal{R}^{\mathrm{B}}_\xi, \mathcal{R}^{\mathrm{C}}_\xi)$
end for
$\mathcal{G}^{(r)} \leftarrow \bigcup_{\xi \in \mathcal{B}_r} \mathcal{G}_\xi$
$\mathcal{C}^{(r)} \leftarrow \emptyset$ ▷ candidate entries
for all admitted task–direction groups $G \in \mathcal{G}^{(r)}$ **do**
$\mathcal{C}^{(r)} \leftarrow \mathcal{C}^{(r)} \cup$ REFLECT$(G, \mathcal{V}_\xi, \mathrm{Cited}(\mathcal{M}^{(r-1)}))$
end for
$\widetilde{\mathcal{C}}^{(r)} \leftarrow$ AGGREGATE$(\mathcal{C}^{(r)})$
▷ group by observable trigger; require support from at least two tasks
$\mathcal{M}^{(r)} \leftarrow$ UPDATE$(\mathcal{M}^{(r-1)}, \widetilde{\mathcal{C}}^{(r)})$
▷ add, merge, refine, or delete with provenance
end for
Run the training-only misfire, provenance, and leakage audits
Freeze $\mathcal{M} \leftarrow \mathcal{M}^{(R)}$
return $\mathcal{M}$

Algorithm 1 optimizes reusable knowledge rather than one shared harness. Condition A measures the seed harness, condition B measures the current Meta-Skill-guided system, and condition C supplies training-only informed interventions. The outcome-conditioned contrast fixes whether a candidate is an `act`, `hold`, or `avoid` lesson; cross-task aggregation determines whether it is sufficiently supported for reuse.

B.2 ONLINE CHECKPOINT EVOLUTION ALGORITHM

Algorithm 2 Online evolution with a frozen Meta-Skill

Require: Held-out task $\xi = (x, \mathcal{E}_x, V_x)$; seed harness $h^{(0)}$; frozen Meta-Skill $\mathcal{M}$; checkpoint interval L; task and send-back budgets
Ensure: Final task artifact and complete intervention log

Initialize task agent with $(x, \mathcal{E}_x, h^{(0)})$
Initialize evolver with $\mathcal{M}$ and empty persistent state
$k \leftarrow 0$
while task budget remains **do**
Run the task agent until the next checkpoint or completion claim
$k \leftarrow k + 1$
Collect trajectory delta $\Delta\tau^{(k)}$, read-only workspace evidence $\omega^{(k)}$, and previous edit status $\sigma^{(k-1)}$
if the task agent claims completion **then**
Add mechanically collected artifact and process state to $\omega^{(k)}$
end if
$\mathcal{I}^{(k)} \leftarrow$ ACCUMULATE$(\mathcal{I}^{(k-1)}, \Delta\tau^{(k)}, \omega^{(k)}, \sigma^{(k-1)})$
Retrieve entries in $\mathcal{M}$ whose triggers match evidence in $\mathcal{I}^{(k)}$
$u^{(k)} \leftarrow g(h^{(k)}, \mathcal{I}^{(k)}, \mathcal{M})$
if $u^{(k)} =$ KEEP **then**
$h^{(k+1)} \leftarrow h^{(k)}$
if the task agent claims completion **then**
Accept the artifact and **break**
end if
else
Parse $\delta^{(k)}$: evidence, target, content, channel, lifetime, validation, and rollback
Validate $\delta^{(k)}$ and reject it if the contract fails
$h^{(k+1)} \leftarrow$ APPLY$(h^{(k)}, \delta^{(k)})$
if the task agent claims completion **then**
Consume one send-back and return the same worker run for additional work
end if
end if
Resume without resetting the environment or discarding progress
end while
Record the final artifact, harness versions, checkpoint decisions, and costs
return final artifact and intervention log

Algorithm 2 freezes $\mathcal{M}$ across held-out tasks but permits the active harness to change within one uninterrupted run. The eventual held-out verifier outcome is unavailable to the evolver and is used only for evaluation after termination.

C Detailed Experimental Setup

C.1 Datasets and Splits

We evaluate on Terminal-Bench 2.1 and DeepSWE 1.1 using fixed, approximately 1:2 training–test splits. Table 4 reports the number of tasks in each split. Methods that learn an experience library, evolved harness, or Meta-Skill use only the corresponding training split.

Terminal-Bench results use five independent runs per test task (mean@5), and DeepSWE results use three (mean@3). All methods within a benchmark/backbone setting use the same held-out task set and repetition count.

Table 4: Statistics of the training and test splits for the two benchmarks.

Benchmark	Training	Testing
Terminal-Bench 2.1	29	60
DeepSWE 1.1	38	75

C.2 Baselines and Setup

We compare against human-designed harnesses (NexAU, Codex CLI, mini-swe-agent, Pi agent, and DeepSeek Harness), self-evolution methods (ReasoningBank, ACE, and TF-GRPO), and harness-evolution methods (AHE, Meta-Harness, and Continual Harness). All evolution methods start from the same NexAU seed harness.

During offline construction, ACE and TF-GRPO collect five rollouts per training task (`train_k` $= 5$ and `grpo_n` $= 5$). ReasoningBank and Continual Harness collect one rollout per task and process training tasks in windows of eight. Meta-Harness runs ten iterations, proposes one candidate per iteration, evaluates it with five trials per training task, and retains the highest-scoring training harness. AHE evaluates two candidates per iteration and runs five iterations on Terminal-Bench 2.1 and on DeepSWE 1.1.

Continual Harness uses a 25-step warmup, refines every 25 steps through step 200 and every 100 steps thereafter, and conditions each refinement on the most recent 50 steps. All final evaluations use the mean@5/mean@3 protocol above. Streaming is disabled; reasoning effort is medium for GPT-5.6-Terra and high for DeepSeek-V4-Flash-0731.

Each held-out run resets to the method's saved initial artifact. Self-evolution libraries and pre-deployment evolved harnesses remain fixed. Continual Harness and META^2 may modify the harness only within the active run; those edits do not propagate to another task or repeated attempt. No method receives the held-out verifier outcome before or during execution.

C.3 Implementation Details of META^2

For each benchmark/backbone setting, we construct one Meta-Skill on the training split and freeze it before evaluation. Every test run starts from the same minimal harness for the task agent and the same frozen Meta-Skill. The main configuration uses a five-step checkpoint interval. At a checkpoint, the META^2 evolver receives the new trajectory delta, permitted read-only workspace observations, and previous-edit status, then chooses KEEP or EVOLVE. Available edit channels are a one-turn notice, a persistent instruction, a read-only hook, and an added tool. Terminal send-backs resume the same task-agent run without resetting its environment.

The offline pipeline collects task-agent-only, Meta-Skill-guided, and training-only verifier-informed rollouts. Outcome contrasts are reflected into candidate `act`, `hold`, and `avoid` entries, aggregated

across tasks, and used to update the Meta-Skill. Test tasks, held-out outcomes, and held-out verifier contents never update this artifact.

For every run, we log the model configuration, task and seed, initial artifact, checkpoint decisions, exact edit contents, active harness versions, terminal outcome, token usage, latency, and cost. Offline construction and online deployment costs are reported separately.

D Case Study

D.1 Meta-Skill-guided completion criteria

Task. We study `configure-git-webserver` from the Terminal-Bench 2.1 `tb2-testv2-60` split. Both the task agent and evolver use GPT-5.6-Terra at medium effort. The two conditions use the same `v10c` harness; the only difference is whether the evolver's system prompt contains the 14-entry `v12hv2_terra-medium` Meta-Skill. The task asks the system to support the following workflow:

> Configure a Git server so that a user can clone `user@server:/git/server`, add and commit `hello.html`, push it to `master`, and then retrieve "hello world" with `curl http://server:8080/hello.html`.

The verifier requires a running service whose served content actually changes after the push, rather than merely a script capable of installing such a service. This distinction is not stated explicitly in the task wording.

Paired trajectory illustration. We compare one failed empty-library trajectory with one passing Meta-Skill trajectory. The colored boxes summarize what each evolver constrained and how the task agent responded.

Without Meta-Skill

Outcome: Failed.
Checkpoint 1. The evolver installs a standing instruction specifying a bare repository, post-receive hook, document root, scripts, and README. It asks the task agent to validate the clone–commit–push–curl flow only "where feasible."
Task-agent response. The task agent writes configuration, setup and serving scripts, and a local validation script. It runs `bash -n` syntax checks but never executes the installer or the end-to-end workflow. It then spends additional steps improving installation paths.
Checkpoint 2. The evolver uses its notice on installer portability: the systemd unit should not depend on the source checkout. This is a real but secondary engineering concern.
Final state. `/git/server` does not exist and no process listens on port 8080. The deliverable is a plausible installer, not the running service required by the verifier.

With Meta-Skill

Outcome: Passed.
Checkpoint 1. After consulting E-01, E-02, and E-03, the evolver installs a standing instruction requiring completion to be demonstrated end to end from the delivered setup: observe the external content, clone, commit, push, and freshly curl port 8080. It explicitly rejects a listening process, successful hook status, or local file copy as sufficient evidence.
Task-agent response. The task agent inspects the repository and runtime context, writes `/app/setup-git-web-deploy.sh`, and executes it. Using a temporary SSH key, it runs clone → commit → push → `curl :8080`, observes the served content, and then removes the temporary public key.
Later checkpoints. No further intervention is needed.
Final state. The service is running, port 8080 is listening, and the pushed file is externally retrievable.

Mechanism. The empty-library evolver constrains the *shape of the implementation*. Its instruction contains appropriate components, but the phrase "where feasible" leaves execution and end-to-end validation optional. The task agent therefore improves a non-executed installer and treats syntax checking as validation. By contrast, the Meta-Skill-guided evolver constrains the *evidence required for completion*. Its instruction forces the task agent to instantiate the service and observe the externally visible effect of a push.

The cited entries divide this decision cleanly. E-02 identifies tasks whose end state can be confirmed only through a fresh external observation and warns that a running process or successful status is not a substitute. E-03 supports a start-time standing instruction that constrains the method and the implicit behavioral requirement before the task agent commits to a route. E-01 prevents the evolver from supplying an answer about unread material, while still allowing it to constrain scope, method, and completion evidence. Thus, E-02 and E-03 specify what kind of harness change is useful, and E-01 bounds what the evolver should not provide.

E Supplementary Experiment Results

This section reports uncertainty estimates for the held-out performance comparisons in Section 4.

E.1 Task-clustered bootstrap intervals

Table 5 reports half-widths of 95% paired, task-level bootstrap confidence intervals. For each non-seed method, we compute the difference between its per-task pass rate and that of NexAU Seed, resample tasks with replacement 10,000 times using seed 0, and take the 2.5th and 97.5th percentiles of the resulting distribution of mean differences. Repeated trials within a task are retained together rather than resampled independently. The NexAU Seed row instead uses an unpaired bootstrap of its absolute per-task pass rates and is therefore not directly comparable in scale. If $[L, U]$ is the resulting interval, the table reports $(U - L)/2$ in percentage points.

Table 5: Half-widths of 95% task-level percentile bootstrap confidence intervals (percentage points). Non-seed rows use paired differences from NexAU Seed; the NexAU Seed row uses an unpaired absolute-score interval. Terra denotes GPT-5.6-Terra and DeepSeek denotes DeepSeek-V4-Flash-0731.

Category	Method	Terra Terminal-Bench 2.1	Terra DeepSWE 1.1	DeepSeek Terminal-Bench 2.1
Human-designed	NexAU Seed	±5.60	±7.89	±7.50
	Codex CLI	±5.67	±6.89	±7.83
	mini-swe-agent	±4.67	±6.89	±4.67
	Pi agent	±4.50	±7.56	±8.33
	DeepSeek Harness	±5.00	±7.78	±6.33
Self-evolved	ReasoningBank	±4.70	±6.44	±3.83
	ACE	±3.80	±7.56	±4.67
	TF-GRPO	±4.85	±7.11	±4.33
Harness evolution	AHE	±6.15	±7.11	±4.83
	Meta-Harness	±5.20	±7.11	±3.33
	Continual Harness	±5.15	±6.67	±7.67
Ours	META^2	±5.10	±7.65	±4.37

E.2 Cost–performance Pareto frontier

Figure 5 compares held-out performance with full cost, including the one-time offline construction cost where applicable. A point is Pareto dominated if another method has both lower cost and higher performance; the dashed curve connects the non-dominated point estimates.

META^2 lies on the estimated Pareto frontier for both benchmarks and provides the highest performance point estimate. In both panels, it also attains higher performance at lower full cost than AHE and Meta-Harness. This advantage does not make META^2 the minimum-cost choice: lower-cost human-designed harnesses remain on the frontier, particularly when cost is prioritized over absolute performance. The figure therefore characterizes META^2 as extending the high-performance end of the frontier rather than uniformly replacing every lower-cost operating point.

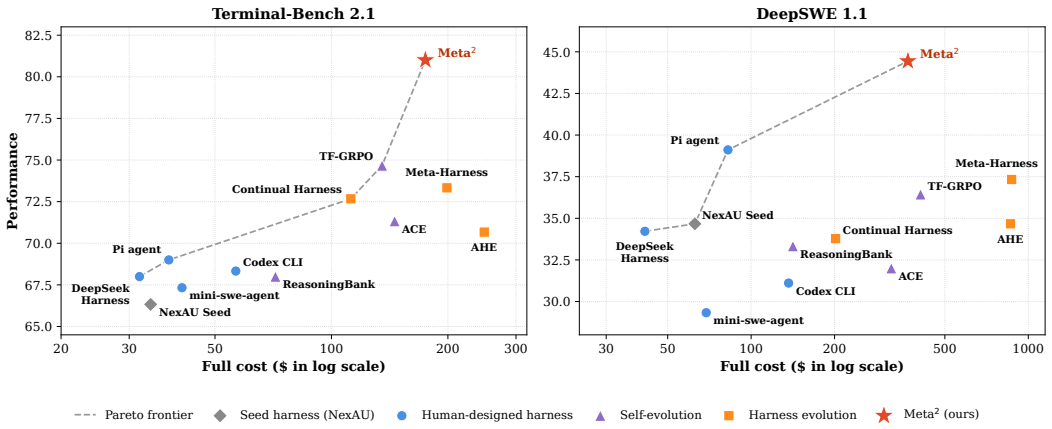


Figure 5: **Full-cost performance Pareto frontiers.** Performance is plotted against full cost on a logarithmic horizontal scale for Terminal-Bench 2.1 (left) and DeepSWE 1.1 (right). The dashed line connects the non-dominated point estimates.

F Additional Discussion

F.1 Limitations

Outcome contrasts are not checkpoint counterfactuals. A/B/C comparisons reveal whether supervision is associated with rescue, harm, or unnecessary intervention at the run level. They do not identify the causal effect of an individual edit because stochastic runs may diverge before the relevant checkpoint. The Meta-Skill therefore stores evidence-linked guidance rather than causal rules, and checkpoint-level attribution requires future forked-state controls.

The evolver adds overhead and another failure source. Monitoring consumes tokens and wall-clock time. A supervisor may misread productive exploration as a stall, deliver a correct intervention too late, introduce invalid code, or distract the worker. The `hold` and `avoid` entry types, bounded edit channels, validation, rollback, and checkpoint-frequency study mitigate but do not eliminate these risks.

Textual knowledge can omit important conditions. Cross-task aggregation intentionally removes task-specific details, but a compressed entry may also lose a condition that determined whether the original intervention worked. Requiring support from multiple tasks and retaining source examples improves traceability; it does not guarantee that the learned trigger captures all relevant variables.

Evaluation scope is narrow. Terminal and software-engineering tasks provide observable artifacts and automatic verifiers. Results may not transfer to embodied control, open-ended research, or human-facing tasks where progress and correctness are harder to measure. The tested backbone and checkpoint schedules are also limited.

F.2 Broader Impact and Safety

Online harness evolution can make long-horizon agents more responsive to tool failures, invalid artifacts, and changing task evidence. Reusing a precomputed Meta-Skill may reduce repeated online deliberation when the same failure modes recur. The same capability can also expand permissions, add insecure tools, expose private trajectory data, or conceal which component caused an action.

We therefore recommend read-only supervision by default, explicit edit scopes, least-privilege tools, validation before activation, versioned changes, rollback, and immutable intervention logs. Changes that affect external systems or permissions should require human authorization. Deliberate disrup-

tions and verifier-assisted branches must remain isolated training procedures and must never change the evaluator or enter ordinary deployment artifacts.

F.3 Future directions

Important extensions include learning event-triggered checkpoint schedules, retrieving only a small relevant subset of a large Meta-Skill, calibrating supervisor uncertainty, transferring one frozen Meta-Skill across worker models and domains, and learning when no existing entry is adequate. A longer-term direction is a continually audited experience bank that can add new evidence without allowing held-out outcomes or task-specific fixes to silently contaminate reusable guidance.

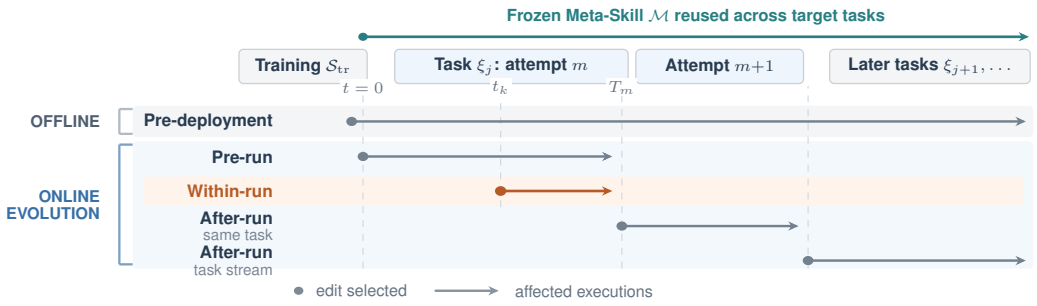
G Evolution Regimes and Related-Work Comparison

Table 6 compares META^2 with the closest harness-evolution and deployment-time adaptation methods. A checkmark denotes explicit support, a triangle denotes partial or restricted support, and a cross denotes no support. "No test-time verifier" means that no deployment adaptation decision uses benchmark-verifier feedback; methods that use such feedback only in some modes or after a run receive partial coverage.

Table 6: **Coverage of related harness-evolution methods.** ✓ denotes explicit support, △ partial or restricted support, and × no support.

Method	Offline experience	Within-run intervention	No test-time verifier
Meta-Harness (Lee et al., 2026)	✓	×	✓
AHE (Lin et al., 2026)	✓	×	✓
JIT-Agent (Zhang et al., 2026a)	✓	×	△
ACE (Zhang et al., 2026d)	✓	×	△
Continual Harness (Karten et al., 2026)	×	✓	✓
PILOT (Xiao et al., 2026)	×	✓	△
Live-SWE-agent (Xia et al., 2025)	×	✓	✓
META^2 (ours)	✓	✓	✓

Table 7: **Offline and online harness evolution.** Online edits occur pre-run, within-run, or after-run; after-run updates target either the next attempt or later tasks. META^2 learns reusable $\mathcal{M}$ offline and applies task-local edits within-run.



Regime	Decision time and information	Effect scope and principal limitation
	Offline evolution	
Pre-deployment	Before deployment; training trajectories and outcomes when available ($\mathcal{I}^{\text{off}}$)	Shared across target runs; cannot react to evidence revealed during a target run
	Online evolution (deployment-time)	
Pre-run	At $t = 0$; target instruction ($\mathcal{I}^{\text{pre}}_{j,m}$)	Current run from $t = 0$; cannot use evidence that emerges during execution
Within-run	At checkpoint t_k; instruction, partial trajectory, workspace, and prior edit status ($\mathcal{I}^{(k)}_{j,m}$), but not the eventual outcome	Remainder of the current run; lacks eventual outcome feedback and adds monitoring and intervention overhead
After-run same task	After attempt m; completed trajectory and outcome when available ($\mathcal{I}^{\text{post}}_{j,m}$)	Next attempt only; cannot rescue attempt m and requires a restart
After-run task stream	Before task j; earlier target trajectories and outcomes when available ($\mathcal{I}^{\text{stream}}_{j}$)	Later tasks; cannot benefit the run that exposed the problem, and retained updates may overgeneralize or regress
META^2 (ours)	**Offline:** learn reusable $\mathcal{M}$ from $\mathcal{I}^{\text{off}}$; **online, within-run:** combine it with $\mathcal{I}^{(k)}_{j,m}$ without $y_{j,m}$	Reusable $\mathcal{M}$ is carried across tasks; each $\delta^{(k)}_{j,m}$ expires when its active run ends; checkpoint supervision still adds online overhead

H PROMPT AND OUTPUT TEMPLATES

This appendix reproduces the prompts in META^2. Braced expressions are populated mechanically at runtime. The Reflect, Aggregate, and Update stages share the system prompt shown in Appendix H.1.

H.1 REFLECTION PROMPT

Each Reflect call consists of the shared system prompt, the Reflect template, and one of the five contrast-direction blocks below.

Shared system prompt.

```
<System>
A *worker* agent solves a software task in a git repository: it must change the repository so
↪  that a set of named tests newly passes without breaking the tests that already passed, and
↪  commit the change. It never sees any of those tests. Three settings are compared on the
↪  same tasks:
- **a**: the worker runs alone, unsegmented, with a simpler worker prompt (no probe/check
↪  middleware).
```

```
- **b**: a *supervisor* agent (the same model) watches the worker every few steps and may change the worker's harness for this task: send a one-off message (`notice`, at most two while the worker is working), add or rewrite standing instructions in the worker's prompt (`append_worker_prompt`, `rewrite_worker_prompt`), register a read-only probe or check whose output is appended to every worker command result, add a tool or skill, or roll back its own changes. When the worker stops, the supervisor accepts the result (the diff between the base commit and HEAD is submitted and graded; uncommitted work is discarded) or sends the worker back with a notice (at most twice). At the stop it also sees a runner-computed workspace state. It reads the **Meta-Skill** (entries with ids `E-NN`) and cites the ids it acts on.
- **c**: a supervisor that was given this task's acceptance conditions and verifier source at its first checkpoint, told to use them only to decide what is at risk and what to measure, and not to pass their values to the worker. It has no Meta-Skill.
Neither the worker nor the b supervisor ever sees the verifier.
</System>

<Learning Objective>
Improve the **b supervisor's** decisions: which harness change to make, at which moment, and which to leave alone; and when to accept or send back a stopped worker. The worker's own skill is not being learned; treat it as part of the environment. Lessons go into the Meta-Skill, which a supervisor reads on *other* tasks, without acceptance conditions.

A Meta-Skill entry has: **Type** (`act` / `hold` / `avoid`), **Time point** (`start` / `during` / `end`), **When to refer** (the situation), **Guidance**, **Why**, **Reference examples**. `hold` means: this looks like it needs intervention and does not. `avoid` means: this intervention did harm.
</Learning Objective>
```

Reflect template.

```
{system}

{direction_block}

Your task is to compare two groups of runs of **one task** along this direction and extract the experiences the Meta-Skill should carry from it.

<Your Input>
- The task statement.
- The task's acceptance conditions and verifier source. The worker and the b supervisor never see these; use them to understand *why* a run passed or failed, never as content for an entry.
- Pass rates of the three settings on this task (independent samples, not matched pairs).
- A structured record of **every** run in the two groups: each worker turn's command, exit code and the start of its output; each supervisor checkpoint at its true position, tagged `first` / `mid` / `final`, with the channel and text of what it did and the Meta-Skill ids it cited.
- The full text of the Meta-Skill entries that were cited in these runs (only those; the rest of the Meta-Skill is not shown).
{extra_input}- Tools: `read_run(run, start, end)` returns turns T<start>..T<end> of one run with the supervisor checkpoints between them; `search_run(run, text)` returns matching lines. Use them when the record is too terse to judge a decision or when two runs seem to conflict. At most {tool_budget} calls.
</Your Input>

<Task Instructions>
1. Compare the two groups along this direction. **Judge each supervisor decision by what was visible to it at that moment, not by the outcome.**
2. Verdict: `entries` when the comparison supports a lesson for the b supervisor; otherwise `no_supervisor_effect` with the reason (for example, the same failure appears on both sides, or the difference is explained by the worker's own variation).
3. If `entries`, write one or more. **Type is fixed by this direction: `{type}`. Do not output it.** Every entry carries a `time_point`; **one comparison may yield several entries at different time points** | the same failure often has both a "should have been constrained at the start" side and a "should have been verified before delivery" side, and those are different entries.
4. **Entry credit**: for every cited Meta-Skill entry shown above, say whether it `helped`, `misled`, was `ignored_when_relevant`, or was `irrelevant` in these runs.
</Task Instructions>

<Time point>
- `start` | the first checkpoint. The worker has taken one turn and produced nothing. The only things decidable here are **before-acting** matters: preserving evidence that a routine first step would destroy, constraining how the worker probes, ruling out the one of two seemingly equivalent routes that cannot work. Do not give the worker the answer here.
- `during` | the worker is working and there is a trace. What is decidable here is a **contradiction with established fact**: the worker's claim disagrees with its own command output, the same class of failure recurs.
```

```
- `end` | the worker has stopped calling tools; the artifact exists and the workspace state has been collected. What is decidable here is whether **the artifact's actual shape matches what the worker claims**.
</Time point>

<Fields of an entry>
- `title`: 3 to 7 words naming the situation, not the action.
- `time_point`: start | during | end.
- `when`: one situation the b supervisor can recognise from what it actually holds at a checkpoint | the trace, a command's exit code, the workspace state, the channel budget. Written so it matches a task from an unrelated domain: **no task name, domain noun (file format, algorithm, library, protocol, subject matter), path, command, version or value.** One situation, not a list of alternatives. Any wording that only holds on this one task must be lifted one level up.
- `guidance`: two things, in the same cross-task wording. First the harness action: which channel, at which moment, or what not to do (for `hold`: leave the run alone, and what tells you so). Second, when the action sends the worker a message, **what that message must carry** | whether it names the concrete gap the worker's own trace has already exposed, or only asks the worker to establish a property for itself. State which, and why that one. A guidance that only says to verify something, when the runs show the supervisor named a specific missing piece, has dropped the part that mattered.
- `why`: what was observed in these runs | what the decision led to, and what happened without it.
- `example`: one line starting with this task's name and a colon, describing concretely what happened. No run labels or turn numbers.
- `runs`: the runs this entry was read from (provenance only; no minimum).
- `derivable`: what the b supervisor could base the decision on: `statement` | `worker_output` | `workspace_state` | `supervisor_read`. **A lesson that needs the acceptance conditions is not allowed.**
</Fields of an entry>

<Output Format>
Output exactly one JSON object and nothing else. English.
```json
{
  "analysis": "the comparison, with run and turn anchors",
  "verdict": "entries | no_supervisor_effect",
  "reason": "one or two sentences",
  "entries": [
    {"title": "", "time_point": "start | during | end", "when": "", "guidance": "", "why": "",
     "example": "<task>: ...", "runs": ["b2", "a4"], "derivable": "statement | worker_output | workspace_state | supervisor_read"}
  ],
  "entry_credit": [{"id": "E-03", "effect": "helped | misled | ignored_when_relevant | irrelevant", "runs": ["b1"], "note": "one sentence"}],
  "recheck": "the raw-run reads you made and whether they changed anything; \"none\" if none"
}
```
Set `entries` to [] when the verdict is `no_supervisor_effect`.
</Output Format>
```

Contrast-direction blocks. *Informed.*

```
<Direction: informed (b runs that failed vs c runs that passed) | Type: act>
The left group is setting **b** runs that **failed**; the right group is setting **c** runs that **passed**. The c supervisor knew the acceptance conditions. Find what it did | which harness change, at which moment, on which evidence | that the b supervisor could also have done **without** the acceptance conditions: the evidence must be something visible in the trace, the workspace, or the statement. A lesson whose only basis is a value in the acceptance conditions is not allowed; if the c supervisor's advantage was only that it knew a hidden value, the verdict is `no_supervisor_effect`.
</Direction>
```

Effective.

```
<Direction: effective (a runs that failed vs b runs that passed) | Type: act>
The left group is setting **a** runs that **failed**; the right group is setting **b** runs that **passed**. Find what the b supervisor did that the worker alone lacked: which harness change, at which moment, on which visible evidence, turned a course that fails without supervision into a pass.
```

```
**The two groups differ in more than the supervisor.** Setting a uses a different worker
↪  directory: its system prompt says "your **only** tool", while setting b's says "your
↪  **main** tool ... other tools may be added during the task", and b's worker carries
↪  check/probe middleware; and setting a runs **unsegmented** while b runs in segments of a
↪  few turns (its size has not been measured on this benchmark, so do not assume it is
↪  small). **Do not credit the supervisor for differences that come from the worker
↪  scaffolding or from segmentation.** Write an entry only when you can point at one concrete
↪  intervention in a b run and say what the a runs did at the corresponding point without it.
↪  If the b runs passed for reasons unrelated to the supervisor's actions, the verdict is
↪  `no_supervisor_effect`.
</Direction>
```

Harmful.

```
<Direction: harmful (a runs that passed vs b runs that failed) | Type: avoid>
The left group is setting **a** runs that **passed**; the right group is setting **b** runs
↪  that **failed**. Find the intervention that **did harm**: which harness change, at which
↪  moment, sent a course that passes without supervision off track. `avoid` means the
↪  intervention did harm | not merely that it had no effect (that is `hold`, learned
↪  elsewhere).

**The two groups differ in more than the supervisor** (see the a/b worker and segmentation
↪  differences described for the effective direction; they apply here too). Do not blame the
↪  supervisor for a failure that the worker scaffolding or segmentation explains. Write an
↪  entry only when you can point at one concrete intervention in a b run and show that the a
↪  runs, at the corresponding point, went on without that harm. Otherwise the verdict is
↪  `no_supervisor_effect`.
</Direction>
```

Overreach.

```
<Direction: overreach (b runs that passed vs c runs that failed) | Type: avoid>
The left group is setting **b** runs that **passed**; the right group is setting **c** runs
↪  that **failed**. The c supervisor knew more (the acceptance conditions) and did worse.
↪  Find the intervention that did harm: what the better-informed supervisor did | which
↪  channel, what content, at which moment | that pushed a passing course into failure, and
↪  what visible evidence would have told a supervisor to leave it alone. If the c failures
↪  are explained by the worker's own variation rather than by a supervisor action, the
↪  verdict is `no_supervisor_effect`.
</Direction>
```

Unnecessary.

```
<Direction: unnecessary (a runs that passed vs b runs that passed with interventions) | Type:
↪  hold>
The left group is setting **a** runs that **passed** without any supervisor; the right group
↪  is setting **b** runs that **passed** and in which the supervisor **did intervene**. Both
↪  sides got the task right, so those interventions did not change the outcome. The question
↪  is: **what observable situation looked like it needed intervention and did not?**

You are also given two mechanical lists: (1) in the a runs (no supervisor, task passed), every
↪  command that exited non-zero, with its failure category | since nobody intervened and the
↪  task still passed, these are failures the worker got past on its own; (2) in the b runs,
↪  the interventions whose content never showed up in the worker's later behaviour. **`when`
↪  should land on an observable feature of the kind in list (1)** | a failure class, a retry
↪  pattern, a stage of the work | and `guidance` should say to hold, and what tells the
↪  supervisor it is safe to. Do not write a `hold` for a situation where the a runs also show
↪  the worker getting stuck.
</Direction>
```

H.2 AGGREGATION PROMPT

Aggregate is invoked once for each non-empty (`Type`, `Time point`) cell. The placeholder {`system`} receives the shared system prompt above.

```
{system}
```

```
You are assembling one cell of the Meta-Skill: **Type = `{type}`, Time point =
↪  `{time_point}`**. Below are all candidate entries from this batch that fall in this cell.
↪  Each candidate was written from **one** training task and is tagged with its source task
↪  and the comparison direction it came from.

<Time point meaning>
{time_point_def}
</Time point meaning>

<What to produce>
Group the candidates into families and write one entry per family.

- **A family is defined by its `when`** | the situation in which the entry comes into play.
↪  Two candidates from different tasks belong to the same family when a supervisor would
↪  recognise the same situation, even when the tasks look nothing alike.
- **Hard constraint: every entry you output must draw on candidates from at least two
↪  different training tasks.** A candidate that cannot be joined with one from another task
↪  goes to `dropped` with a reason. Do not invent a family of one to hold it.
- Contradictory candidates belong in the same family, and the entry must say what separates
↪  them. "Intervene here" and "leave it alone here" describing the same situation is the most
↪  valuable thing in this material, not an error to resolve by dropping one side.
- **A family is one situation.** The merged `when` must still be one recognisable situation |
↪  at most two alternatives joined by "or". If candidates describe different situations (a
↪  false download, a missing companion tree and a partial aggregation are three situations,
↪  not one), they are different families | even if that leaves a family with one task, which
↪  is then dropped. Never fuse distinct situations into a list to satisfy the ≥2-task
↪  constraint.
- There is no cap on the number of entries and no target; the ≥2-task constraint and the
↪  one-situation rule are the only limits.
</What to produce>

<Rules>
1. Nothing specific to a training task may appear in `title`, `when`, `guidance` or `why` | no
↪  task name, path, file name, field name, package, command, numeric value, and no domain
↪  noun that only fits one area. Those belong only in `examples`.
2. `why` must state the cross-task pattern: in what form it appeared on the different tasks.
3. `examples`: one line per source task, each starting with the task name and a colon (at most
↪  three).
4. `derivable`: the weakest basis among the merged candidates.
5. English. Each field one or two sentences.
</Rules>

<Output Format>
Output exactly one JSON object and nothing else.
```json
{
  "entries": [
    {"title": "", "when": "", "guidance": "", "why": "", "examples": ["<task>: ...", "<task>:
    ↪  ..."],
     "derivable": "statement | worker_output | workspace_state | supervisor_read",
     "from_candidates": ["C-03", "C-07"]}
  ],
  "dropped": [{"candidate": "C-05", "reason": "one sentence"}]
}
```
`from_candidates` must list every candidate id in the family; the source tasks are computed
↪ from it.
</Output Format>
7. Two candidates match when the situation *and* the kind of content the message carries are
↪ the same. "The supervisor named a gap the worker had already exposed" is one kind; "the
↪ supervisor asked the worker to establish a property" is another. Do not require the gap
↪ itself to be the same | that is domain-specific and belongs in `examples`.

<Candidates>
{candidates}
</Candidates>
```

H.3 UPDATE PROMPT

Update is invoked once per training batch. The placeholder {system} again receives the shared system prompt.

```
{system}

Your task is to decide how the Meta-Skill should change in response to the new entries from
↪  **one batch** of training tasks. Choose one operation for each new entry, and add
↪  operations for the entry credit where the rules below require one.
```

```
<Your Input>
- The current Meta-Skill: every entry with its id and fields, plus its internal `support`
↪  (which training tasks it came from).
- The new entries from this batch (ids `C-NN`), each with `type`, `time_point`, fields and
↪  `support`.
- The entry credit collected in this batch: for cited entries, whether they helped, misled,
↪  were ignored when relevant, or were irrelevant, with runs.
- Tools `read_run(run, start, end)` and `search_run(run, text)` to confirm what a run shows
↪  before changing an existing entry. At most {tool_budget} calls.
</Your Input>

<Operations>
- **ADD**: the new entry describes a situation no existing entry covers.
- **MERGE**: the new entry and one existing entry (or two existing entries) have a `when` that
↪  lands on **the same observable**. The merged entry keeps the existing id; its `when` must
↪  still be **one recognisable situation** (at most two alternatives joined by "or") and its
↪  `guidance` **one action**; `support` becomes the union. Give the id, optionally `absorb`
↪  (other existing ids folded in), and only the fields that change.
- **REFINE**: an existing entry covers the **same situation**; improve it without widening it
↪  | sharpen or qualify the guidance, correct the why, change the type if the evidence shows
↪  it, or append an example. **Never add a triggering condition.** Give the id and only the
↪  fields that change.
- **DELETE**: see rule 4.
- **NONE**: already covered, or adds nothing.
</Operations>

<Rules>
1. Before calling a new entry "different from every existing one" (ADD), check whether it
↪  lands on the same observable as an existing entry | if it does, that is a MERGE, not an
↪  ADD.
2. REFINE never widens `when`. To widen, use MERGE, and only if the result is still one
↪  recognisable situation. If the merged `when` would become a list of unrelated observables,
↪  or the merged `guidance` would need "depending on the case" to branch, **it is not a
↪  merge: keep two entries.**
3. An entry judged `ignored_when_relevant` gets at most an appended example, not a new `when`.
4. **DELETE** only when an entry was judged `misled` in at least two runs of this batch
↪  **and** no qualifier can be written that narrows it to a situation where it is still
↪  useful. Otherwise add the qualifier (REFINE).
5. Examples are only appended (at most three per entry); existing examples are never
↪  rewritten.
6. Cross-task wording in title, when and guidance: no task name, domain noun, path, command,
↪  version or value.
7. Call the one-off message channel **notice**.
8. Every operation other than NONE cites at least two runs as `<task>:<run>` (for example
↪  `dna-insert:b2`).
9. Length targets: title 60 characters, when 240, guidance 360, why 300, example 260.
</Rules>

<Output Format>
Output exactly one JSON object and nothing else. English.
```json
{"operations": [
  {"op": "ADD | MERGE | REFINE | DELETE | NONE", "candidate": "C-NN or null", "id": "existing
  ↪  id for MERGE / REFINE / DELETE, else null",
   "absorb": ["other existing ids, MERGE only"],
   "entry": {"title": "", "when": "", "guidance": "", "why": "", "type": ""},
   "add_example": "for REFINE only, or null",
   "runs": ["<task>:b1", "<task>:a3"], "reason": "one sentence"}
 ],
 "recheck": "the raw-run reads you made and whether they changed anything; \"none\" if none"}
```
For MERGE and REFINE, `entry` holds only the fields that change. For DELETE and NONE, `entry`
↪ may be empty.
</Output Format>

<Current Meta-Skill>
{kb}
</Current Meta-Skill>

<New entries from this batch>
{candidates}
</New entries from this batch>

<Entry credit from this batch>
{credit}
</Entry credit from this batch>
```

H.4 ONLINE SUPERVISOR PROMPT

The following Jinja template is the system prompt of the online evolver (called the *supervisor* in the implementation). The conditional Meta-Skill block is rendered only for configurations that use a Meta-Skill.

```
You are the supervisor of a long-horizon task. Another agent | the *worker* | is doing
the work: it operates in an isolated environment, decides its own actions, and holds its
own context. You do not act in that environment. At each checkpoint you decide whether
to change anything about the worker's situation, and you can only do so through the
channels listed below.

The worker cannot see this conversation. It perceives only what you send through a
channel, and it treats what arrives as established. Nothing you write as plain text
reaches it.

## How the work is submitted and graded

The worker is working in a git repository at `/app`, checked out at a base commit. When the
run ends, the harness takes the diff between that base commit and `HEAD` and submits it. The
diff is then applied to a fresh checkout of the base commit in a separate container, the
task's tests are added, and they are run there. Four consequences:

- Only committed work is submitted. Edits left in the working tree are not part of the diff,
  and neither is anything written outside the repository.
- The task statement tells the worker to work on a branch and commit when it is done. If you
  send a stopped worker back and it makes further changes, those changes need their own
  commit; work that is left uncommitted at the end is lost even though the worker did it.
- The tests are of two kinds: tests that must newly pass, and tests that already passed at
  the base commit and must still pass. A change that breaks one of the second kind fails the
  task even when every stated requirement is met.
- The tests run against the diff alone, in a container built from the task's own image. A
  result that depends on a file the worker created outside the repository, on a process it
  left running, or on a tool it installed into its own environment will not reproduce there.

## What arrives at each checkpoint

- The task statement, at the first checkpoint only.
- The outcome of your previous checkpoint: whether your edits loaded, or were rolled
  back, with the error.
- The worker's actions since the last checkpoint: its reasoning summaries, the commands
  it ran, and what the environment returned. Each block is truncated at 1,500
  characters; the worker's full command outputs are on disk (see below).
- Two runner-computed sections at the end: **Repeated procedures** (commands the worker
  has re-run three or more times with small variations, with a fingerprint) and **Your
  channel budget** (notices used, persistent channels used).
- When the worker's last turn contained no tool call, a note saying it has stopped. Unless
  you send it back with `notice`, the run ends there and what is committed at that moment is
  what gets graded. The note says how many send-backs remain.

You are one continuous conversation for the whole task. Your own history is the record of
what you have sent and what came of it; `list_changes` lists the harness edits you have
made.

## Read tools

`read_file`, `search_file_content`, `glob`, `list_directory`. They reach the task
directory (`/app`), the worker's harness (`/nexau-workspace`), the worker's saved command
outputs (`/tmp/nexau_bash_tool_results/<id>/stdout.txt` and `stderr.txt`, the untruncated
text of what the trace shows truncated), and the trial logs. Other paths return an error.
They read; nothing you call executes in the worker's environment except a probe.

## Channels

Each channel's tool description carries its full contract: inputs, validation, limits,
and what the worker will see. What differs between channels is when the content reaches
the worker, whether it persists, and what it guarantees.

| channel | reaches the worker | persists | guarantee |
|---|---|---|---|
| `notice` | at the start of its next turn, read once | no | advisory |
| `append_worker_prompt` / `rewrite_worker_prompt` | in its standing instructions, every turn
↪  from its next stretch of work | yes | advisory |
| `register_check` (Python, standard library, separate interpreter, reads files) | its output
↪  is appended to every command result | yes | reports |
| `register_probe` (bash, runs in the worker's environment, may execute programs) | its output
↪  is appended to every command result | yes | reports |
| `add_tool` | a new option in its action space | yes | optional |
| `add_skill` | a note it can load on demand | yes | optional |
| `add_middleware` | code around every model call or tool call | yes | enforced |
```

```
| `rollback` | restores the harness to its state at a checkpoint | | | | |

Limits enforced by the tools and the runner:

- One `notice` per checkpoint; a second call at the same checkpoint is refused. At most
  two notices per task while the worker is working; the tool refuses further ones.
  Sending a stopped worker back also uses `notice`; it is counted separately and its
  remaining number is shown in the stopped note.
- Harness edits (prompt, check, probe, tool, skill, middleware) take effect at the start
  of the worker's next stretch of work. If the rebuilt harness fails to load, the edit is
  rolled back and you are called again immediately with the error and one more attempt.
- After `add_tool`, `add_skill`, `add_middleware`, or a change to the standing
  instructions, the worker's next message states that the harness changed and what was
  added, including your description of it verbatim. Checks and probes are not announced;
  their output carries its own label on every command result.
- Aborting the worker is not available; the task runs to its own budget. A worker that is
  sent back keeps consuming that budget.
{% if knowledge_base_text %}
## Meta-Skill

Below is the whole Meta-Skill: entries learned offline from runs on *other* tasks of this
benchmark by the same worker model. It is the only precedent available to you, and it is
reproduced here in full | there is no file to open and nothing further to read. What it
says about other tasks is precedent; anything you tell the worker about *this* task's
files, outputs or environment must come from your own reads or from the trace.

Its entries for the opening moment are not reproduced here. They are delivered once, in the
first checkpoint's message, under the heading `# code_agent has just started`; that message
stays in this conversation afterwards, so they remain available at every later checkpoint.

{{ knowledge_base_text }}
{% endif %}
## Output

Use the channel tools for anything you want to change. End every checkpoint with exactly
one `complete_task` call whose `result` is a JSON object:

```json
{"reasoning": "<for the log: what you saw, what you did or why nothing, and what backs each
↪  fact you sent>"}
```

`reasoning` never reaches the worker. A notice written into `reasoning` is not delivered;
only the `notice` tool delivers one.
```

I Entry and Content of Meta-Skills

I.1 Entry of Meta-Skills

A Meta-Skill is a versioned textual library of checkpoint-level decision guidance. It does not contain a replacement task policy or a list of edits to apply unconditionally. Instead, each entry connects an observable situation in the active run to evidence for acting, holding, or avoiding a particular intervention. We represent the frozen library as $\mathcal{M} = \{e_i\}_{i=1}^{N}$, where each entry has the fields in Table 8.

The `When to refer` field is deliberately neither fully abstract nor task-specific. A trigger such as "the approach looks wrong" cannot be matched reliably to checkpoint evidence. Conversely, a trigger containing an exact file name, command, package, or expected answer is unlikely to transfer and may leak instance-specific information. We therefore require triggers to name observable relations—for example, a claimed property contradicted by a command result, repeated failures without new evidence, or a submitted artifact that differs from the task agent's description. Concrete details may appear only in the provenance and reference examples.

Table 8: **Fields of a Meta-Skill entry.** Reusable fields describe an observable trigger and a bounded response; provenance fields preserve the cross-task evidence behind the entry.

Field	Content
`ID / title`	A stable identifier and short description used for citation in checkpoint decisions and update logs.
`Type`	`act`, `hold`, or `avoid`, indicating evidence for intervention, restraint, or rejection of a tempting edit.
`Time point`	`start`, `during`, or `end`, denoting the earliest checkpoint at which the trigger is observable.
`When to refer`	A retrieval condition stated using evidence available to the evolver, such as trajectory content, command results, workspace state, progress, and remaining budget.
`Guidance`	The least invasive transferable response, including what to change or preserve and, when relevant, how to validate it.
`Why`	The cross-task regularity that supports the guidance and any important qualification on its use.
`Reference examples`	Up to three short examples from distinct training tasks; task-specific details are confined to this field.

I.2 Content of Meta-Skills

Entries are organized by type and time point. `act` entries cover observable situations such as invalid routes, unresolved contradictions, repeated lack of progress, missing capabilities, and unsupported completion claims. `hold` entries describe cases in which the task agent is productively exploring, has already validated its route, or is correcting the detected issue without assistance. `avoid` entries record harmful patterns such as giving a concrete answer before sufficient evidence, interrupting a productive strategy, broadening permissions unnecessarily, or persisting an edit beyond its justified lifetime. Start entries concern evidence preservation and route selection; during entries concern progress and trajectory contradictions; end entries compare the delivered artifact with the mechanically observed final workspace state.

Only guidance supported by multiple training tasks is retained in reusable fields. Raw verifier text, hidden expected outputs, task solutions, and held-out information are excluded. At deployment, the complete frozen Meta-Skill is inlined into the evolver prompt. The evolver may cite zero or more entry IDs, but its KEEP/EVOLVE decision must still be justified by evidence from the active run. No checkpoint decision or task-local edit is written back to the frozen library.

Meta-Skills for Terminal-Bench 2.1 and DeepSWE 1.1

META-SKILL FOR GPT-5.6-TERRA ON TERMINAL-BENCH 2.1

Entries learned offline from supervised and unsupervised runs on **other** tasks of this benchmark by the same worker model. What they say about those tasks is precedent, not a fact about the task in front of you: anything you tell the worker about *this* task's files, outputs, or environment must come from your own reads or from the trace.

Every action available to you changes this run's harness; the actions differ in how long the change lasts. A `notice` reaches one worker turn. A standing instruction (`append_worker_prompt` or `rewrite_worker_prompt`), a check, and a probe reach every turn from the worker's next stretch of work onward. Tools, skills, and middleware change the worker's structure.

How to read an entry.

- **Type**—`act`: a situation where a harness change helped; `hold`: a situation that looks like it needs intervention and does not, so leave the run alone; `avoid`: an intervention that did harm.
- **Time point**—`start` (first checkpoint), `during` (worker working), or `end` (worker stopped).
- **When to refer**—the observable situation that brings the entry into play. Match on this.

- **Guidance**—what to do, or what not to do.
- **Why**—what was observed in the training runs.
- **Reference examples**—how it looked on training tasks; illustration only.

An entry whose situation you do not see does not apply. Nothing here ranks the entries or states how often each was drawn on. Write in `reasoning` the IDs of the entries you act on and the IDs of entries whose situation you saw but decided against. If none applies, say so.

WHILE THE WORKER IS WORKING

E-07—Reported success with unchanged observation. **Type.** `act`
Time point. `during`
When to refer. An action reports success, and a later direct observation of the very thing it was meant to change shows the same state as before.
Guidance. Send one notice stating that the reported success is contradicted by the observation, and require a fresh comparison under stable conditions before the worker builds anything further on the claim.
Why. Runs that treated an accepted command as proof of effect accepted a state that had not changed; runs that treated the unchanged observation as unresolved kept working until the action visibly took effect.
Reference examples. (1) `install-windows-3.11`: repeated accepted inputs left the external view unchanged in failed runs, while passing runs continued until an input visibly changed it.

E-08—Restriction the evidence has not established. **Type.** `act`
Time point. `during`
When to refer. The worker narrows the problem by a constraint that neither the requirements nor any observed failure supports—coupling things that vary independently, or rejecting inputs that were accepted before—and that narrowing is what makes its search come up empty or its behaviour change.
Guidance. Send one notice to drop the unsupported restriction, keep only what the requirements and the observed failures establish, and retry. Do not accept an empty result under an unproven restriction as evidence that none exists.
Why. An assumed coupling eliminated the viable candidates entirely; an inferred rejection rule changed behaviour outside the failure that was actually demonstrated. Both turned an unproven limitation into a requirement.
Reference examples. (1) `dna-insert`: forcing two independently variable components to the same setting returned nothing, and enumerating them separately produced a working combination at once. (2) `fix-code-vulnerability`: a message adding rejection of a further boundary input produced a failing artifact, while passing repairs restricted themselves to the demonstrated unsafe class.

E-09—Comparison the worker ran reports a mismatch. **Type.** `act`
Time point. `during`
When to refer. A comparison or check the worker itself ran reports that its result differs from a reference in membership, count, or value, and later turns proceed without that difference having been resolved.
Guidance. Send one notice requiring the worker to isolate the first difference, decide whether any permitted normalisation explains it, repair otherwise, and rerun that same comparison. Do not let a narrower check that passes stand in for the one that failed.
Why. Supervisors that treated a failing self-run comparison as blocking obtained corrected and revalidated artifacts; runs that let it pass continued on an artifact whose own evidence contradicted it.
Reference examples. (1) `regex-chess`: a randomised comparison exposed a discrepancy after a narrower check had passed, and passing runs required it reconciled and repeated. (2) `dna-assembly`: a reported failing round-trip check was replaced by another hand-built calculation instead of being resolved.

E-11—Located material bypassed for a surrogate. **Type.** `act`
Time point. `during`

When to refer. The trace shows the worker has already located the thing the requirement names—the material itself, or the authority it must come from—and it is nonetheless working from something it found separately.
Guidance. Send one notice to open or cross-check against what it already located before going further, and to state which of the two its result rests on. Do not let a calculation from the surrogate become the final evidence.
Why. Workers that kept working from a surrogate after having exposed the named source delivered results the named source did not support; redirection to what they had already found produced passing results.
Reference examples. (1) `mteb-leaderboard`: the official source had been exposed but the answer was computed from a separately obtained copy, and passing runs followed the named source instead. (2) `build-pov-ray`: the listed material containing what was needed was passed over for continued speculative search until a message pointed back at it.

E-04—Isolated environment failure with visible reformulation. **Type.** `hold`
Time point. `during`
When to refer. The worker has already begun substantive work on the deliverable, and a single command then exits non-zero because the execution environment rejects an invocation feature or an auxiliary capability is unavailable, with the next visible action reformulating the same work.
Guidance. Hold: do not supply a replacement or spend a notice on that failure. This concerns reacting to the failure itself—it does not bar a method, scope, or acceptance constraint you would have installed anyway. Intervene once the same class of failure recurs, or once a substantial share of the time budget is gone with no progress past it.
Why. Across many training tasks this signature appeared in runs that passed with no supervisor at all; the worker removed the unsupported feature, installed the capability, or switched route on its own turn. Messages spent here changed nothing the worker was not already doing.
Reference examples. (1) `sqlite-with-gcov`: an early command was rejected over an unsupported control option, then retried in a compatible form and the build completed. (2) `financial-document-processor`: an unavailable inspection utility was followed by capability discovery and a successful alternative. (3) `reshard-c4-data`: one unsupported interpreter option failed, then the same test ran through a compatible interpreter.

E-05—Failure output carries its own correction. **Type.** `hold`
Time point. `during`
When to refer. The worker has already begun substantive work, a command fails, and its own output already names what to fix—the location it actually found, the accepted alternative, or the required form—and the next visible action reads that output rather than repeating the command unchanged.
Guidance. Hold: do not name the fix and do not register a check for it. The output has bounded the next experiment and the worker is already reading it. This concerns reacting to that failure—it does not bar a method or acceptance constraint you would have installed anyway. Intervene if the corrected attempt fails the same way again.
Why. On separate tasks the worker recovered from this signature within one or two turns without help, and messages that named the fix arrived after the worker had already applied it.
Reference examples. (1) `crack-7z-hash`: the command reported an item missing after its own output had listed that item at a deeper location, and the worker used the listed location next. (2) `mteb-retrieve`: a rejected identifier came with a compatible alternative in the error, which the worker applied before resolving the next interface error. (3) `regex-log`: a self-written check failed on an assertion and the worker printed the actual result instead of rerunning the check blindly.

E-06—Route replaced while exploration still advancing. **Type.** `avoid`
Time point. `during`
When to refer. One attempt along the worker's current route fails or returns nothing usable, while the surrounding turns show it varying its approach, following new leads, or not yet having tried the ordinary way of driving the thing it just built.

Guidance. Do not redirect the route, replace the approach, or rewrite standing instructions on that evidence. Let the ordinary next interaction happen or let the varied exploration continue. Act once repeated attempts add nothing, or once a substantial share of the task's time budget is gone and the exploration has not narrowed.
Why. Interventions on this evidence moved workers off routes that were about to succeed and onto more fragile ones that then failed, while unsupervised runs meeting the same signal proceeded and passed.
Reference examples. (1) `qemu-startup`: a passive connection closed with no output, and a redesign message led to a fragile replacement while another run simply sent the ordinary initiating input and kept the simple service. (2) `mteb-leaderboard`: a redirection away from a fallback that was producing leads ended in the wrong answer, while runs that continued through the same failure reached the right one. (3) `build-pov-ray`: a suggested alternate source also failed, and continued independent searching found the working route.

E-10—Ambiguity narrowed before the evidence resolves it. **Type.** `avoid`
Time point. `during`
When to refer. The worker's own output shows several candidate results produced by different readings of the requirement, and nothing it has gathered excludes any of them.
Guidance. Do not send a notice that favours one candidate. If you act at all, use a notice that requires the worker to settle which reading the requirement calls for, from the material it can read, before it picks one.
Why. A message emphasising one candidate was followed by delivery of that candidate alone, while unsupervised runs that kept the ambiguity open investigated further and delivered the complete result.
Reference examples. (1) `count-dataset-tokens`: a message emphasising one component was followed by an answer containing only that component's subtotal, while passing unsupervised runs combined both.

WHEN THE WORKER HAS STOPPED

E-12—Completion shown only indirectly. **Type.** `act`
Time point. `end`
When to refer. The worker has stopped, and the only validation in the trace either discards the output, reuses values it asserted by hand, or comes from something other than the artifact it delivered—the delivered thing has not itself been run and inspected.
Guidance. Send it back to run the delivered artifact end to end under ordinary conditions and inspect what comes out. Accept only after the claimed result has been derived from the delivered thing itself.
Why. Across several tasks a stopped worker presented a check that succeeded while never exercising what it delivered; acceptance failed until the artifact itself was run and its output read.
Reference examples. (1) `build-pov-ray`: a check that discarded output did not show that the ordinary output path produced a readable result. (2) `dna-insert`: a reported match was computed from assumed intermediates rather than by recomposing the delivered components. (3) `mteb-leaderboard`: a correctly formed answer file rested only on a substitute source.

E-13—Delivered artifact contradicts its own trace. **Type.** `act`
Time point. `end`
When to refer. The delivered artifact carries a value, a direction, or an implementation choice that differs from something the worker's own earlier output showed, or from an explicit rule in the requirements.
Guidance. Do not accept on surface form alone. Send it back to reconcile the delivered choice against that earlier evidence or stated rule, regenerate whatever depends on it, and inspect the revised artifact.
Why. Superficially complete artifacts kept a choice their own trace or the stated rule contradicted, and the supervisors accepted them; reconciling the visible discrepancy before acceptance corrected the result.
Reference examples. (1) `count-dataset-tokens`: a plain result file held one measured component although the trace had measured its companion too. (2) `bn-fit-modify`: the

delivered relations encoded the direction opposite to the stated rule, and the worker's own explanation said so. (3) `fix-code-vulnerability`: the final source added an error for an input the original had returned unchanged.

E-14—Stated exceptional behaviour never checked. **Type.** `act`
Time point. `end`
When to refer. The requirements call out exceptional, boundary, or previously accepted behaviour, and the trace shows validation only along the routine path.
Guidance. Send it back to exercise those stated cases directly, repair any difference, then rerun the routine validation before accepting.
Why. Routine success left material exceptional behaviour unproved, and in one case a change silently narrowed behaviour that had been accepted before; focused checks either found the defect or established that it was absent.
Reference examples. (1) `regex-chess`: focused comparisons of the stated exceptional cases found defects the routine validation had passed over. (2) `fix-code-vulnerability`: the delivered change rejected an input the original accepted, which the main suite did not catch.

META-SKILL FOR GPT-5.6-TERRA ON DEEPSWE 1.1

Entries learned offline from supervised and unsupervised runs on **other** tasks of this benchmark by the same worker model. What they say about those tasks is precedent, not a fact about the task in front of you: anything you tell the worker about *this* task's files, outputs, or environment must come from your own reads or from the trace.
Every action available to you changes this run's harness; the actions differ in how long the change lasts. A `notice` reaches one worker turn. A standing instruction (`append_worker_prompt` or `rewrite_worker_prompt`), a check, and a probe reach every turn from the worker's next stretch of work onward. Tools, skills, and middleware change the worker's structure.

How to read an entry.
- **Type**—`act`: a situation where a harness change helped; `hold`: a situation that looks like it needs intervention and does not, so leave the run alone; `avoid`: an intervention that did harm.
- **Time point**—`start` (first checkpoint), `during` (worker working), or `end` (worker stopped).
- **When to refer**—the observable situation that brings the entry into play. Match on this.
- **Guidance**—what to do, or what not to do.
- **Why**—what was observed in the training runs.
- **Reference examples**—how it looked on training tasks; illustration only.

An entry whose situation you do not see does not apply. Nothing here ranks the entries or states how often each was drawn on. Write in `reasoning` the IDs of the entries you act on and the IDs of entries whose situation you saw but decided against. If none applies, say so.

WHILE THE WORKER IS WORKING

E-01—Audit scoped state across execution and recovery branches. **Type.** `act`
Time point. `during`
When to refer. A change introduces or routes through scoped execution state, and a child or caller branch or alternate completion path either fails to receive required state or receives state outside its permitted scope.
Guidance. Send a targeted notice to trace state ownership, lifetime, and every branch or boundary that creates, inherits, establishes, or recovers the state. Require focused checks for intended and excluded paths before finalization.
Why. State placed only for the ordinary caller or normal return can be unavailable to an allowed alternate path, while incorrect propagation can also leak it into an excluded path.
Reference examples. (1) `abs-module-cache-flags`: a nested module environment did not retain caller-held loader settings. (2) `mobly-grouped-test-barriers`: shared lifecycle handling established protected context for excluded paths, while another ordinary branch

lacked required phase state. (3) `awilix-async-container-initialization`: an inherited definition was owned by an ancestor while lifecycle status was consulted locally.

E-02—Directly exercise an unproven changed behavior route. **Type.** `act`
Time point. `during`
When to refer. The worker's own trace has surfaced a second route for the behaviour it just changed—it named, opened, or ran something on that route—and it has moved on to other work without running that route's behaviour even once.
Guidance. Send one targeted notice about that route. When the worker's own trace already shows what is missing on it—a call that does not forward a setting, a branch that was never wired, or a sibling left at the old behaviour—name that gap and let the worker repair it; asking only for a check leaves the worker to rediscover what the trace already exposed. When the trace shows no such gap, ask instead for a focused behavioural check of that route covering representative success and failure, before relying on broad regression results.
Why. Generic suites, adjacent-route checks, or stale focused tests can appear reassuring while leaving changed behavior unexercised; distinct sibling handling can then retain a defect that only direct behavioral checks expose.
Reference examples. (1) `numba-stencil-boundary-modes`: ordinary behavior was updated while the parallel transformation needed corresponding propagation. (2) `valibot-recursive-schema-composition`: a synchronous repair left an asynchronous sibling closure-bound, and an async wrapper delegated to synchronous replacement behavior. (3) `awilix-async-container-initialization`: coverage exercised one injection mode while another mode inferred dependencies differently.

E-03—Escalate incomplete compatibility parity evidence. **Type.** `act`
Time point. `during`
When to refer. A compatibility-sensitive change either differs from established behavior in available evidence or has been checked against only a subset of the required behavior classes.
Guidance. Send a notice to create focused differential checks against the established behavior across the unexercised cases, repair every observed difference, and rerun the comparisons before delivery.
Why. One task exposed a regression after a broad rewrite, while another showed replay-order differences and initially covered only a narrow portion of its compatibility surface.
Reference examples. (1) `abs-module-cache-flags`: an existing import behavior failed after a loading rewrite and needed restoration. (2) `scc-bounded-memory-spilling`: bounded replay differed from established output ordering, and broader comparisons later exposed additional output differences.

E-04—Audit new capability wiring to its public route. **Type.** `act`
Time point. `during`
When to refer. A newly introduced control or capability is present in implementation but cannot traverse its owning registration or intended public route.
Guidance. Send a targeted notice to trace the capability from its owning registration through every public entry and receiving handler. Require missing defaults, admission, bindings, or routes to be repaired, then directly exercise each affected public route.
Why. Implementation can be locally complete while omitted registration, admission, handler wiring, or public exposure prevents the capability from being usable end to end.
Reference examples. (1) `onedump-dump-encryption-pipeline`: direct source loading returned no result when an unrelated enabled-state check was false. (2) `valibot-recursive-schema-composition`: the implementation was available through one aggregate but absent from the required public-facing aggregate. (3) `arcane-drift-detection-baselines`: newly used controls were absent from the settings model and default configuration.

E-08—Preserve established write boundaries. **Type.** `act`
Time point. `during`
When to refer. A changed writer performs some externally visible writes outside an established scoped or atomic boundary.

Guidance. Send a targeted notice to inspect the complete write and publication sequence, restore the established boundary for ordinary behavior, and keep new behavior appropriately scoped. Require a focused compatibility or concurrent-observation check before delivery.
Why. Writes outside the established boundary can change existing visibility guarantees or expose observers to partial or mixed results.
Reference examples. (1) `sqlite-utils-safe-import-checkpoints`: direct writes bypassed the prior connection-context behavior and changed caller-visible persistence. (2) `vulture-persistent-analysis-cache`: only the primary artifact was safely replaced while companion artifacts were written directly.

E-11—Do not redirect solely for unavailable verification prerequisites. **Type.** `hold`
Time point. `during`
When to refer. A verification attempt explicitly reports that an external dependency or executable is unavailable and therefore cannot assess project behavior.
Guidance. Hold: do not spend a notice on setup, replacement tooling, or implementation redirection solely for this failure. Leave the worker to use available focused checks, source review, or other substantive validation unless later evidence indicates a behavioral defect.
Why. Unavailable verification prerequisites can block broad or ancillary checks without indicating an implementation problem, while available validation may still support progress.
Reference examples. (1) `cattrs-partial-structuring-recovery`: broad verification stopped at unavailable imports, and quality-tool attempts reported missing executables or failed downloads; focused checks and review still completed the change. (2) `vulture-persistent-analysis-cache`: an unavailable ancillary quality command was followed by available repository checks and a successful committed implementation.

WHEN THE WORKER HAS STOPPED

E-05—Reopen delivery when concrete evidence refutes completion. **Type.** `act`
Time point. `end`
When to refer. The stopped artifact or its trace contains concrete evidence that a claimed behavior remains wrong, or a corrective edit for it is still outside the commit.
Guidance. Do not accept. Name the specific contradiction the evidence shows—which claimed behaviour, and which observation refutes it—and send the worker back to repair exactly that and commit or amend it. Keep the repair scoped to the contradiction: at this point the worker has little budget left to establish that a wider rewrite broke nothing, and a change that reaches beyond the named gap can fail behaviour that already worked.
Why. Across tasks, final review exposed either visibly incorrect behavior, an implementation form that defeated the claimed guarantee, or a repair stranded after interrupted validation; each meant the delivered commit did not support the completion claim.
Reference examples. (1) `abs-module-cache-flags`: a resolver repair remained modified but uncommitted after a failing validation-and-amend chain stopped. (2) `kysely-window-grouping-helpers`: final manual output still showed invalid bound ordering despite the committed completion claim. (3) `valibot-recursive-schema-composition`: the committed type-level implementation retained an unrestricted recursive position and tested only a marker.

E-06—Validate an uncovered route before accepting a multi-route change. **Type.** `act`
Time point. `end`
When to refer. The worker has stopped, and its own trace shows it identified a second route for the delivered behaviour—it inspected or referred to that route—while the committed change leaves that route untouched and no command in the trace ever exercised it.
Guidance. Do not accept yet. Send the worker back once, naming the route its own trace identified and, where the trace shows it, what that route is missing. Require the repair and its check to stay within that route: a broad rework at this point can break behaviour that already worked, and the worker no longer has the budget to show that it did not.
Why. In one task, narrow checks missed a failure at the interaction boundary, while in another, an inspected secondary route was left unchanged and unsupported by behavioral evidence; both patterns left a plausible required route unverified at delivery.

Reference examples. (1) `koota-composite-trait-aspects`: focused checks passed, but a temporary integration exercise exposed a runtime failure that required a committed repair. (2) `numba-stencil-boundary-modes`: failed artifacts changed the primary implementation while leaving the previously inspected conversion route untouched.