HARNESSTTT: LEARN TO OPTIMIZE HARNESS AT TEST-TIME FOR SCIENTIFIC DISCOVERY

Anonymous authors
Paper under double-blind review

ABSTRACT

Scientific discovery with language models requires finding the best valid solution within a finite computational budget. We study how feedback from this search can improve the procedure used to conduct it. We introduce HARNESSTTT, a test-time learning framework that trains a harness proposer while keeping the executor frozen. The proposer generates executable harnesses comprising system prompts, reusable skills, tools, middleware, and execution controls. Each harness guides a budgeted discovery rollout, whose outcome provides feedback for updating the proposer. A discovery-oriented policy-gradient objective combines gap-normalized rewards, leave-one-out advantages (RLOO), and max-sharpening to favor high-performing proposals. Generated components are validated before execution, and the search retains the best valid solution and selected harnesses across rounds. Across thirteen discovery tasks in mathematics, systems, heuristic programming, astrodynamics, and biology, HARNESSTTT with a frozen 9B executor improves on OpenEvolve with the same executor on all eleven benchmark tasks (e.g., from 1.17 to 2.64 on circle packing), improves on the discovery-fine-tuned Finch-9B by up to 92%, holds the best open-weight sub-10B result on nine, and reaches 0.380919 on Erdős minimum overlap, better than the best human reference of 0.380927. A frozen 27B executor improves on the 9B executor on eleven of thirteen tasks, and transferring the learned harnesses to larger executors matches or exceeds the strongest reported system of any backbone on eight of thirteen tasks. Together, these results show that harness search can itself be learned at test time: a trained proposer lets a small frozen executor search effectively, and the harnesses it finds carry over to stronger executors.

1 INTRODUCTION

Scientific discovery advances knowledge and technology by finding solutions beyond what established methods can achieve (Novikov et al., 2025; Romera-Paredes et al., 2024). Recent work (Wang et al., 2025; Ye et al., 2026; Yuksekgonul et al., 2026) on AI for scientific discovery formulates each task as an exploration for the best valid solution to a single challenging scientific problem within a finite computational budget, with the hope of surpassing the best-known results from the best human expert (Georgiev et al., 2025). Success is therefore measured by the best solution found during the exploration, rather than the average quality of all attempts.

Large language models have tackled these tasks through two representative routes as shown in Figure 1. TTT-Discover (Yuksekgonul et al., 2026) performs reinforcement learning at test time (Akyürek et al., 2025; Zuo et al., 2025), directly updating the executor—the model that solves the task—from its search outcomes guided by the reward. This adapts the model to solve problem directly, but requires repeated sampling and training, with 25,600 rollouts per problem in its reported configuration (50 rounds $\times$ 512 rollouts). Moreover, the learned behavioral adaptation remains encoded in the executor's parameters, making individual changes difficult to isolate and limiting their direct transfer to a different or stronger executor. Artifact evolution methods (Lange et al., 2025; Novikov et al., 2025; Romera-Paredes et al., 2024; Sharma, 2025; Ye et al., 2026) instead keep the executor and harness fixed – adaptation occurs through the feedback artifacts supplied in context. Artifacts are inspectable and reusable, allowing subsequent attempts to build on previous discoveries. However, adaptation within the search occurs through the artifacts supplied in context: the model must infer useful revisions based on its in-context capabilities (Madaan et al., 2023; Shinn et al., 2023), without parameter updates from these outcomes.

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.

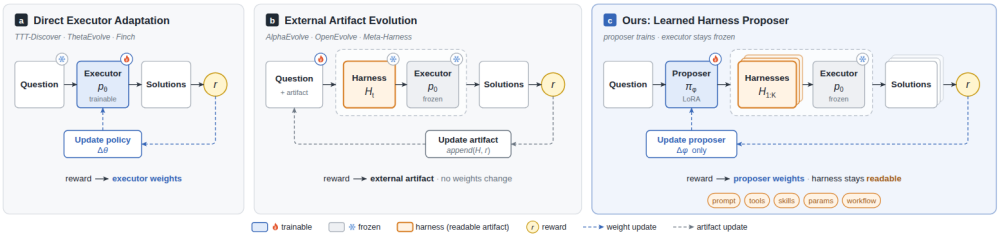


Figure 1: Where is test-time reward retained? (a) *Executor weights*: the policy that solves the task is also the policy being trained (Wang et al., 2025; Yuksekgonul et al., 2026). **(b)** *Proposer context*: the executor stays frozen and a fixed proposer re-reads an archive or harness accumulated over rounds (Lee et al., 2026a; Novikov et al., 2025; Sharma, 2025). **(c)** *Proposer weights (ours)*: a separate proposer is trained to write complete harnesses for a frozen executor, so adaptation is amortized into weights while remaining expressed as inspectable artifacts.

In long-horizon discovery, successive rounds produce increasingly extensive histories. When more of this history is incorporated into the input, accumulating more experience therefore does not reliably translate into more effective exploration (Hsieh et al., 2024; Liu et al., 2024b).

To address the above limitations, we introduce HARNESSTTT, which update the proposer's parameters to generate and refine task-specific harnesses while keeping the executor frozen. Our approach takes inspiration from human researchers, who improve not only the solutions they explore, but also the harness through which they explore them. During the exploration, humans may develop reusable problem-solving skills, revise construction methods, build specialized tools, or reorganize the experimental workflow to improve exploration efficiency. On a circle-packing problem, for example, a researcher might adopt a boundary-first construction strategy, write tools that check pairwise overlaps and boundary violations or compute the largest radii a fixed set of centers admits, and organize geometric diagnostics to guide subsequent refinements. In an agentic system, these changes correspond to adapting the *complete harness* that governs the executor's behavior (Lee et al., 2026a; Weng, 2026; Yang et al., 2024b). These harness-level adaptations shape an entire subsequent sequence of attempts, rather than only the next candidate solution.

HARNESSTTT makes such harness adaptation a learned process as shown in Figure 2. At each round, different from existing methods where the harness only contains only part of the harness: a prompt or context playbook (Agrawal et al., 2025; Khattab et al., 2024; Yang et al., 2024a; Zhang et al., 2026c), a code patch installed at predefined lifecycle hooks of a fixed runtime (Shao et al., 2026), or a fixed four-module protocol (memory, planning, action, and tool/skill orchestration) that draws on a given tool and skill registry (Zhang et al., 2026a), our proposer model generates *complete harnesses* comprising system prompts, skills, tools, workflow, and execution-control parameters. Each generated harness guides the frozen executor through a budgeted discovery rollout from a shared incumbent solution. Improvements in the best valid solution provide rewards, which update the proposer's parameters via our carefully designed reinforcement learning methods, while only a small length of the selected best solution and harness are retained for the next round. Across rounds, discovery experience changes the proposer's policy itself, enabling reliable improvement via learning. Meanwhile, the resulting harness remains explicit and separate from the executor's weights: it can be inspected, revised, and deployed with a stronger executor, providing a mechanism for weak-to-strong transfer (Burns et al., 2024).

To implement HARNESSTTT, it presents two challenges. First, *harness-level credit assignment is indirect and noisy*. The proposer generates a harness rather than a final solution, so its reward is observed only through a separate, stochastic executor rollout. Even when a fixed solution has a deterministic score, the same harness can produce different trajectories and yield different best solutions. The observed reward therefore reflects both harness quality and executor sampling variability. We develop a discovery-oriented RLOO-based update (Ahmadian et al., 2024; Kool et al., 2019) with gap-normalized rewards, leave-one-out advantages without variance normalization (Liu et al., 2025), best-of-K sharpening (Balashankar et al., 2025; Sessa et al., 2025), and filtering of uninformative groups (Yu et al., 2025), designed to limit noise amplification while accommodating heterogeneous reward scales and best-of-group selection. Second, *harness optimization must guard against reward hacking and information leakage*. Higher evaluator scores do not necessarily reflect better solutions: an agent may exploit flaws in the evaluation process (Krakovna et al., 2020; METR, 2025) or use information unavailable under the intended task protocol (Kapoor and Narayanan, 2023). We therefore design a new critic component to address it.

We evaluate HARNESSTTT on thirteen discovery problems spanning mathematical constructions (Georgiev et al., 2025; Novikov et al., 2025), systems benchmarks (Cheng et al., 2025), heuristic programming contests (AtCoder Inc., 2025; Imajuku et al., 2025), astrodynamics, and biology. With a frozen Qwen3.5-9B executor (Qwen Team, 2026a), HARNESSTTT improves on OpenEvolve with the same executor on all eleven benchmark tasks and holds the best open-weight sub-10B result on nine, ahead of the discovery-fine-tuned Finch-9B (Lee et al., 2026b) on every task by up to 91.7%. Scaling the frozen executor to Qwen3.8-27B (Qwen Team, 2026b) improves on *eleven* of thirteen tasks, and transferring the learned harnesses to larger executors matches or exceeds the strongest reported system of any backbone on *eight* of thirteen tasks and comes within 1% on four more. Our ablations show where the gains come from: at matched rollouts, routing reward into the proposer's weights reaches better results earlier than updating the executor or the proposer's context; wider search per round yields late breakthroughs; and our RLOO objective with max-sharpening outperforms standard GRPO and a frozen proposer. Together, these results show that the search procedure itself can be learned at test time with a small frozen model and carried over to stronger executors.

2 RELATED WORK

Self-evolution by adapting the model. One family improves an agent by changing model weights. Test-time training and test-time RL adapt weights per instance or per problem (Akyürek et al., 2025; Muennighoff et al., 2025; Zuo et al., 2025); for discovery, TTT-Discover (Yuksekgonul et al., 2026) and ThetaEvolve (Wang et al., 2025) fine-tune the executor on its own search rollouts and hold strong published $\leq$10B records, while discovery fine-tuning amortizes the skill across tasks by training on evolution-trajectory corpora, as in the Finch models we use as a parameter-matched comparator (Lee et al., 2026b). Closest to us, HASE (Luo et al., 2026) and SIA (Hebbar et al., 2026) co-evolve weights *and* harness, which couples the improver to the improved and requires update access to the executor; HARNESSTTT freezes the executor and trains a separate proposer policy.

Self-evolution by editing the harness. A second family keeps the model frozen and revises an external artifact. Evolutionary program search around frozen LLMs produces state-of-the-art discovery programs: FunSearch (Romera-Paredes et al., 2024), AlphaEvolve (Georgiev et al., 2025; Novikov et al., 2025), open reimplementations and successors (Lange et al., 2025; Sharma, 2025), heuristic evolution (Liu et al., 2024a), equation discovery (Shojaee et al., 2025), reward design (Ma et al., 2024), and multi-agent and meta-evolution systems (Liu et al., 2026; Qu et al., 2026), evaluated on algorithm-engineering suites (Imajuku et al., 2025) and the systems problems of Cheng et al. (2025); research agents automate the surrounding pipeline (Lu et al., 2024). A parallel line lets a frozen model rewrite its own code or scaffold: STOP (Zelikman et al., 2024), the Darwin Gödel Machine (Zhang et al., 2025a), self-improving coding agents (Robeyns et al., 2025; Yin et al., 2025), capability-synthesizing generalists (Qiu et al., 2025), and a survey of the space (Gao et al., 2025). A third optimizes the text or structure around the model: prompt search (Fernando et al., 2024; Guo et al., 2024; Suzgun and Kalai, 2024; Yang et al., 2024a; Zhou et al., 2023), prompt compilation (Khattab et al., 2024; Opsahl-Ong et al., 2024), textual gradients (Cheng et al., 2024; Yuksekgonul et al., 2025), reflective evolution (Agrawal et al., 2025), automated agent and workflow design (Hu et al., 2025; Shang et al., 2025; Zhang et al., 2025b; Zhou et al., 2024), whole-harness editing (Lee et al., 2026a; Zhang et al., 2026b), and in-context self-correction (Madaan et al., 2023; Shinn et al., 2023; Wang et al., 2024a). Across this family the proposer's *weights are fixed*, so design knowledge is re-derived per task rather than accumulated. We train the proposer across tasks and rounds, and its action space includes executable tool code: unlike tool creation that abstracts functions from solution traces (Cai et al., 2024; Qian et al., 2023; Wang et al., 2024b; Yuan et al., 2024), the tools here are emitted by a *learned* policy and admitted through a leak-limiting review chain (§3.2).

3 HARNESSTTT

HARNESSTTT learns the discovery procedure by training a harness proposer while keeping the executor frozen. The proposer specifies how the executor searches through proposed harnesses; the executor conducts budgeted roll-outs to generate candidate solutions and interacts with evaluators to obtain scores for them. Feedback from these evaluators updates the proposer and the search state retained between rounds. We first formulate the discovery objective, then describe the harness representation, proposer update, iterative search, and validation protocol.

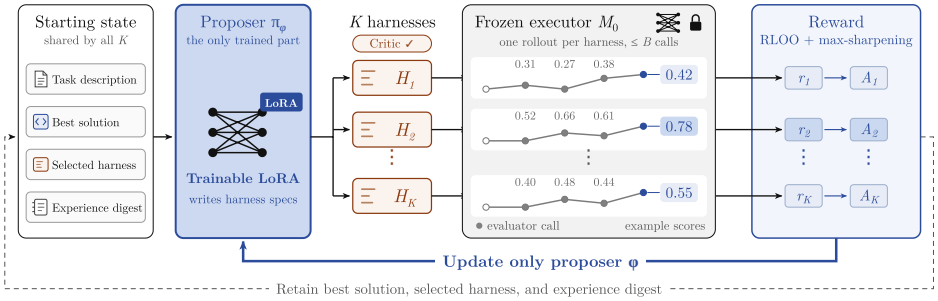


Figure 2: Overview of HARNESSTTT. A trainable proposer generates K candidate harnesses from a shared task state. Accepted harnesses guide a frozen executor through budgeted discovery rollouts. Their outcomes provide group-relative rewards for updating the proposer; selected solutions, harnesses, and an experience digest condition the next round. Executor parameters remain fixed throughout.

Problem formulation and framework overview. A task τ provides a description d_τ, a seed program x_0 with a marked editable region, and a black-box evaluator m_τ; the seed program and evaluator are taken from Lee et al. (2026b). Each rollout may call the evaluator at most B times. An agent pairs a frozen language-model executor M_0 with an executable harness H that governs how M_0 interacts with the task. Each outer round starts from a search state: the incumbent program $x^\star$ (the best found so far, initially x_0) with score $b_\tau = m_\tau(x^\star)$, the incumbent harness $H^\star$, and an experience digest D.

Given this context, the proposer π_ϕ samples K harness specifications. Each round keeps a single incumbent harness $H^\star$, but proposals are dealt round-robin between $H^\star$ and one diversity-preserving runner-up parent: the best candidate within 2% of the top score that adds a new component, or else the second-highest scorer. This keeps components from non-winning candidates alive for another round (Agrawal et al., 2025; Zhang et al., 2025a). In the first round all K proposals share the initial harness. Each specification is merged into its parent, validated, and compiled into a candidate harness H_k; all candidates start from the same program $x^\star$. Only the proposer's LoRA adapter ϕ is trained (Hu et al., 2022); M_0 is never updated. A rollout of M_0 under H_k edits the program and calls the evaluator on programs $x_{k,1}, \ldots, x_{k,T}$ ($T \leq B$); its score is the best valid score it finds, $s_k = \max_{t \leq T} m_\tau(x_{k,t})$, where invalid programs are skipped and a rollout without any valid program counts as failed. The round keeps the best solution found, so the proposer's objective is the best score in the group, never below the incumbent's: $J_K(\phi) = \mathbb{E}_{\pi_\phi}[\max(b_\tau, s_1, \ldots, s_K)]$. The update in §3.1 is a group-relative surrogate that favors high-scoring proposals, not an unbiased gradient of J_K.

Executable harness representation. What the proposer can learn is bounded by what a harness can express. Existing systems represent it narrowly: prompt optimizers rewrite instructions (Agrawal et al., 2025; Fernando et al., 2024; Khattab et al., 2024), reflective agents append notes to the context (Shinn et al., 2023; Zhang et al., 2026c), and Harness-R1 patches a fixed runtime without adding executable tools (Shao et al., 2026). These change what the executor is told but not what it can do, such as adding an analysis tool or restarting a stalled search. We instead represent the harness as an executable program built on NexAU (Nex-AGI Team, 2026) with five editable components (Figure 6; Appendix C): a *system prompt* for strategy and workflow, *skills* for reusable procedures, *tools* including proposer-written Python functions, *middleware* hooks that run between executor steps (e.g., budget reminders, restarts after stalls), and *control parameters* for sampling and iteration limits. The proposer edits a private copy of $H^\star$ and cannot touch the task program, evaluator, executor weights, or shared runtime; unchanged fields are inherited, so a partial specification still yields a complete harness. Each episode ends with `validate_harness`, which checks schemas, mounts, and generated code (§3.2), and `submit_harness`; accepted specifications are compiled and hashed to remove duplicates within the group. The result is a standalone artifact that can be inspected or run with another frozen executor (§4.4).

3.1 LEARNING THE HARNESS PROPOSER

A harness is judged only through executor rollout. Even with a deterministic evaluator, the same harness can lead a stochastic executor to different solutions. An observed score therefore mixes the quality of a proposed procedure with downstream sampling variability. Our RLOO-based update (Ahmadian et al., 2024; Kool et al., 2019) must assign credit under this indirect feedback

while accounting for shrinking absolute gains and an objective that retains the best discovery rather than the average attempt.

Reward. As search approaches a strong reference, meaningful improvements can become small in absolute units. With an unnormalized leave-one-out advantage, these smaller score differences carry less policy-gradient weight even when they close the same fraction of the remaining gap. A valid candidate is rewarded by how much its rollout score s_k improves on the incumbent score b_τ, relative to the remaining gap to a published reference score u_τ (listed in the appendix), or to $|b_\tau|$ once the incumbent has reached it:

$$g_k = \frac{s_k - b_\tau}{Z_\tau}, \qquad Z_\tau = \begin{cases} u_\tau - b_\tau, & b_\tau < u_\tau - \epsilon, \\ |b_\tau| + \epsilon, & \text{otherwise}, \end{cases} \tag{1}$$

A small remaining gap can make a large gain produce an extreme normalized reward. We smoothly cap positive credit below the reference, while retaining graded penalties for rollouts that underperform the incumbent:

$$r_k = \begin{cases} 3\tanh(g_k/3), & b_\tau < u_\tau - \epsilon,\ g_k > 0, \\ \max\{-1,\ g_k\}, & b_\tau < u_\tau - \epsilon,\ g_k \le 0, \\ \mathrm{clip}(g_k, -1, 5), & \text{otherwise}, \end{cases} \tag{2}$$

with $\epsilon = 10^{-9}$. Gains are smoothly capped, losses are linear and floored at -1, and matching the incumbent earns zero; invalid, failed, or repaired candidates receive -1. The reference u_τ is used only for the reward and is never shown to the executor or written into a harness.

Proposals inherit different harness parents, so a cross-parent comparison could reward an inherited advantage rather than the proposed revision. we compute advantages within groups $\mathcal{G}$ sharing a parent and prompt: one group of size $K_g = K$ in the first round, then two groups of size $K_g = K/2$. Within each group, valid rewards spanning less than $\delta = 10^{-3}$ are replaced by their mean to avoid crediting near-tied outcomes. Denoting the resulting rewards by $\widetilde{r}_k$, we compute the leave-one-out advantage $A_k^{\mathrm{LOO}} = \widetilde{r}_k - \frac{1}{K_g - 1}\sum_{j \in \mathcal{G}\setminus\{k\}} \widetilde{r}_j$. We omit division by the group's reward standard deviation, which can turn small, potentially noisy differences into order-one advantages (Liu et al., 2025; Shao et al., 2024). The RLOO term thus retains the task-relative magnitude established by gap normalization. All-valid tied groups are excluded from training (Yu et al., 2025); groups receiving identical status penalties are assigned zero advantages and skipped before computing a valid-reward range or temperature. Mixed groups preserve the contrast between tied valid rewards and status penalties. To additionally favor the strongest outcomes retained by the discovery objective, we augment RLOO with a centered softmax sharpening term (Balashankar et al., 2025; Sessa et al., 2025):

$$A_k = (1-\alpha)A_k^{\mathrm{LOO}} + \alpha\left[\mathrm{softmax}(\widetilde{\mathbf{r}}_{\mathcal{G}}/T_{\mathcal{G}})_k - \frac{1}{K_g}\right], \quad T_{\mathcal{G}} = \max\{\Delta_{\mathcal{G}}/4, \epsilon\}, \tag{3}$$

where $\alpha = 0.3$, $\widetilde{\mathbf{r}}_{\mathcal{G}}$ contains the group's processed rewards, and $\Delta_{\mathcal{G}}$ is its valid-reward range. The two terms serve complementary purposes: RLOO preserves the magnitude of relative progress, while range-scaled sharpening emphasizes within-group ranking without reducing feedback to a hard winner label. Near-tie processing suppresses differences below the threshold, but does not distinguish genuine gains from executor noise. The combined advantage is a discovery-oriented heuristic, not an unbiased gradient estimator like standard GRPO (DeepSeek-AI, 2024).

Proposer update. The executor generates the solutions, but the proposer generates the action being optimized. We therefore apply A_k only to the proposer-generated tokens in episode k, masking tool observations and all executor tokens. With $\rho_{k,t}(\phi)$ the ratio of current to sampling-policy token probabilities and $\mathcal{T}_k$ the retained token set, we minimize the clipped surrogate (Schulman et al., 2017) with a KL penalty toward the frozen base proposer:

$$\mathcal{L}(\phi) = -\frac{1}{N}\sum_k \sum_{t \in \mathcal{T}_k} \min\big(\rho_{k,t}A_k,\ \mathrm{clip}(\rho_{k,t}, 1-\epsilon_c, 1+\epsilon_c)A_k\big) + \beta\,\mathcal{L}_{\mathrm{KL}}, \tag{4}$$

$N = \sum_k |\mathcal{T}_k|$, ϵ_c is the framework default, and $\mathcal{L}_{\mathrm{KL}}$ is the low-variance estimator $\pi_{\mathrm{ref}}/\pi_\phi - \log(\pi_{\mathrm{ref}}/\pi_\phi) - 1$ with $\beta = 0.1$. Clipping and the KL anchor constrain the change made from a small batch of indirect feedback. Zero-advantage episodes are omitted; if no training tokens remain, the optimizer step is skipped without interrupting state retention. The persistent rank-64 LoRA adapter

Table 1: HarnessTTT-9B leads the compared open-weight (≤10B) systems on nine of eleven tasks and improves on Finch-9B on all eleven. Bold: best open-weight value per column; underline: second; *Best*: column wins. Proprietary rows are context only; the last row is the relative change over Finch-9B in the metric's direction. † Obtained with a different evolution pipeline and shown for reference only.[1]

Method	Erdős↓	AC1↓	AC2↑	CP26↑	Hadamard↑	AHC039↑	AHC058 (M)↑	EPLB↑	PRISM↑	LLM-SQL↑	Txn↑	Best
Initial program	0.495056	1.518600	0.855800	0.959764	0.143275	534,850	0.00	0.1265	21.89	0.6856	2824.86	
OpenEvolve, proprietary executors (reference only)												
Claude Opus 4.6	0.381880	–	–	2.629300	–	–	–	0.1270	26.26	0.7160	3774.00	
Gemini 3 Pro	–	–	–	2.541400	–	–	–	0.1272	26.24	0.7258	4273.50	
GPT-5	–	–	–	2.541400	–	–	–	0.1272	26.23	0.7155	4237.30	
Open-weight executors (≤10B)												
Qwen3.5-2B	0.381737	1.518600	0.864600	1.253056	0.478009	546,078	0.00	0.1265	21.89	0.6856	2832.86	0/11
Finch-2B	0.381346	1.518400	0.892000	1.535134	0.400476	545,256	329.36	0.1269	22.26	0.6860	2949.85	0/11
Qwen3.5-4B	0.416924	1.518600	0.880200	1.680787	0.384332	542,077	0.00	0.1266	21.89	0.6856	2732.24	0/11
Finch-4B	0.386460	1.517300	0.893300	1.806808	0.146199	551,844	331.47	0.1267	22.87	0.6857	**4761.90**	1/11
Qwen3.5-9B	0.385512	1.518600	0.880100	1.172702	0.397184	553,582	134.49	0.1269	22.36	0.6858	3584.23	0/11
Finch-9B	0.381100	1.514100	0.912200	1.936000	0.480585	553,759	525.29	0.1265	23.93	0.7024	3636.36	0/11
TTT-Discover† (Qwen3-8B)	0.380932	1.505250	**0.947200**	**2.635983**	–	–	–	–	–	–	–	2/11
ThetaEvolve† (R1-Qwen3-8B)	–	1.506803	0.946900	**2.635983**	0.576400	–	–	–	–	–	–	1/11
Nano-Discover† (Qwen3-8B)	0.380935	1.505395	0.945766	**2.635983**	–	–	–	–	–	–	–	1/11
HarnessTTT-9B (ours)	**0.380919**	**1.504680**	0.933900	**2.635983**	**0.921053**	**559,534**	**713.55**	**0.1272**	**26.26**	**0.7415**	4255.32	**9/11**
Δ *vs. Finch-9B (%)*	+0.05	+0.62	+2.38	+36.2	+91.7	+1.04	+35.8	+0.55	+9.74	+5.57	+17.0	

is trained for three epochs per round with learning rate 10^{-5} and batch size eight. Episodes are truncated to 18k tokens while preserving the final submission; ϵ_c follows the training framework's configured clipping threshold. The updated adapter then proposes the next round's harnesses, closing the learning loop without changing M_0.

3.2 Experience retention and harness validation

Incumbent and bounded context. At the end of each round, the best valid program in the group and its harness become the new incumbent (Figure 2), and the next round starts from them rather than from the seed. Solution quality therefore accumulates, and rewards measure progress beyond the current starting point. Under a fixed evaluator the incumbent score never decreases, but this follows from retention and does not by itself show that the proposer improved. Unlike reflective methods (Agrawal et al., 2025; Shinn et al., 2023), we do not feed the proposer the history of earlier candidates. Such a history grows every round, raises cost, and buries informative outcomes in a long context that models use poorly (Liu et al., 2024b). Feedback that drifts across rounds would also make within-group reward differences reflect context rather than proposer quality. The proposer instead reads a fixed-size *experience digest*, rebuilt each round: the incumbent harness and program (truncated), the seed and current best scores, and a stall flag. Experience across rounds is thus carried by the digest and the proposer's weights. Proposals from the same parent share this digest and differ only in sampling seed.

Harness validation. Evaluator scores cannot tell real improvements from exploits of the evaluation interface (Baker et al., 2025; Denison et al., 2024; Krakovna et al., 2020; Skalse et al., 2022; Zhong et al., 2026), so every generated tool is validated before rollouts. The *critic* in Figure 2 is this validation pipeline, not a learned value function. Tools access the task only through a restricted interface that withholds the evaluator implementation, ground truth, and network access, and they must pass static checks and a sandboxed self-test. A failing tool gets at most two repair attempts, screened for evaluator or target-score content, and is dropped if it still fails. One violation in human-curated notes is disclosed and excluded from performance claims (§4.2). These controls describe the evaluated protocol, not an independently verified guarantee.

4 Experiments

4.1 Evaluation Design

Task suites. The parameter-matched comparison uses the eleven tasks for which Finch-9B reports results (Lee et al., 2026b): five *mathematical construction* tasks (**Erdős**, **AC1**, **AC2**, **CP26**, **Hadamard**) (Georgiev et al., 2025; Novikov et al., 2025; Sharma, 2025); two *heuristic-contest programming* tasks (**AHC039**, **AHC058**) scored under the official AtCoder 150-case protocol (AtCoder

[1]We follow the default evolution pipeline of Sky-Discover: https://github.com/skydiscover-ai/skydiscover.

Table 2: Each stage of proposer learning improves the harness on most tasks, and a larger executor adds a further step on ten of thirteen. The 9B rows differ only in how the proposer is updated. Each Δ row is the relative change over the row above it, in the metric's direction.

Condition	Erdős↓	AC1↓	AC2↑	CP26↑	Hadamard↑	AHC039↑	AHC058↑	EPLB↑	PRISM↑	LLM-SQL↑	Txn↑	V2↓	scRNA↑
Initial, 9B	0.456591	1.518200	0.896100	1.477767	0.360961	557,225	134.49	0.1265	24.02	0.6856	3610.11	3.839529	0.641
Context, 9B	0.395849	1.509800	0.926300	2.370463	0.517212	559,534	562.82	0.1278	23.04	0.7267	3906.25	3.458023	0.701
Δ (%)	+13.3	+0.55	+3.37	+60.4	+43.3	+0.41	+318.5	+1.03	−4.08	+6.00	+8.20	+9.94	+9.36
HarnessTTT-9B	0.380919	1.504680	0.933900	**2.635983**	**0.921053**	559,534	713.55	0.1272	**26.26**	0.7415	4255.32	3.446415	0.709
Δ (%)	+3.77	+0.34	+0.82	+11.2	+78.1	0.00	+26.8	−0.47	+14.0	+2.04	+8.94	+0.34	+1.14
HarnessTTT-27B	**0.380902**	**1.503778**	**0.934868**	**2.635983**	0.806604	**564,457**	**841.39**	**0.1285**	**26.26**	**0.9676**	**4370.00**	**3.430206**	**0.714**
Δ (%)	+0.004	+0.06	+0.10	0.00	−12.4	+0.88	+17.9	+1.02	0.00	+30.5	+2.69	+0.47	+0.71

Table 3: With harness transfer, HarnessTTT matches or exceeds the reported frontier on eight of thirteen tasks and is within 1% on four more. Baseline columns are the strongest reported system per task family regardless of backbone. For HarnessTTT, the first column searches with its own frozen executor and the second moves the learned harness to GPT-5.5 and continues search. **Bold**: state of the art per task; underline: second best.

	Reported systems										Our method			
Task	OpenEvolve	AlphaEvolve	Hyra	SimpleTES	TTT-Discover	ShinkaEvolve	CORAL	ALE-Agent	GEPA	LEVI	HarnessTTT (9B)		HarnessTTT (27B)	
backbone	*multiple*	*Gemini 2.0*	*Hy3*	*gpt-oss-120B*	*gpt-oss-120B*	*multiple*	*Opus 4.6*	*Gemini/ GPT*	*GPT-5/ Gem. 3*	*mixed*	*Qwen3.5-9B*	*transfer → GPT-5.5*	*Qwen3.8-27B*	*transfer → GPT-5.5*
Mathematical discovery														
Erdős ↓	0.380910	0.380924	**0.380859**	0.380868	0.380875	0.38156	0.38089	–	–	–	0.380919	0.380923	0.380902	0.380868
AC1 ↓	1.506604	1.503170	**1.502850**	1.503871	1.502863	–	–	–	–	–	1.504680	1.506805	1.503778	1.503871
AC2 ↑	0.944900	0.961020	0.962901	0.962694	0.959180	–	–	–	–	–	0.933900	0.934210	0.934868	**0.963589**
CP26 ↑	2.635983	2.635983	2.635983	2.635983	–	2.635983	**2.6360**	–	–	–	2.635983	2.635983	2.635983	2.635983
Hadamard ↑	–	–	–	**0.935673**	–	–	–	–	–	–	0.921053	**0.935673**	0.806604	**0.935673**
Algorithm engineering														
AHC039 ↑	–	–	–	567,503	567,062	558,026	–	550,647	–	–	559,534	**571,450**	564,457	569,409
AHC058 (M) ↑	–	–	–	849.33	848.41	–	–	848.37	–	–	713.55	849.68	841.39	**850.80**
System performance														
EPLB ↑	0.1272	–	–	–	–	0.1290	0.1490	–	0.1445	**0.1523**	0.1272	0.1294	0.1285	0.1288
PRISM ↑	**26.26**	–	–	–	–	**26.26**	**26.26**	–	26.23	**26.26**	**26.26**	**26.26**	**26.26**	**26.26**
LLM-SQL ↑	0.7258	–	–	–	–	0.7240	0.7310	–	0.7134	0.7985	0.7415	0.7375	**0.9676**	0.9672
Txn ↑	4,273.50	–	–	–	–	4,329.00	4,566.00	–	3,984.10	4,464.29	4,255.32	**4,651.16**	4,370.00	4,273.50
Astrodynamics and biology														
V2 ↓	–	–	–	3.430214	–	–	–	–	–	–	3.446415	3.432584	3.430206	**3.430204**
scRNA ↑	0.710	–	–	**0.740**	0.730	–	–	–	–	–	0.709	0.740	0.714	0.736

Inc., 2025; Imajuku et al., 2025); and four *systems algorithm design* tasks (**EPLB**, **PRISM**, **LLM-SQL**, **Txn**) from the ADRS suite (Cheng et al., 2025). The extended suite adds one astrodynamics task (**V2**) and one biological denoising task (**scRNA**), giving the thirteen rows of Table 3 (Ye et al., 2026; Yuksekgonul et al., 2026); these two have no external comparator under the open-model scale rule, and LLM-SQL and Txn are shown for disclosure and reported with caveats.

Evaluation protocol. Following the default setting of Lee et al. (2026b), every task provides a description, a seed program with a delimited editable region, a black-box evaluator, and a hard per-rollout evaluator-call budget. Tables report native metrics and mark their optimization direction. The configured executor interface exposes neither evaluator code nor ground truth (§3.2).

Implementation and reporting unit. The principal executor is frozen Qwen3.5-9B (Qwen Team, 2026a) served with vLLM (Kwon et al., 2023); the scale comparison uses frozen Qwen3.8-27B (Qwen Team, 2026b). The standard configuration runs 20 outer rounds and samples $K{=}32$ (Qwen3.5-9B) or $K{=}8$ (Qwen3.8-27B) harnesses per round; for each harness, the executor runs a single solution trajectory and may call the evaluator at most $B{=}20$ times within it; the proposer is a persistent LoRA adapter (Hu et al., 2022) trained with our proposed advantage (Eq. 3) for three epochs per step at batch size 8 under a KL anchor. Tables report the selected valid outcome of each single search campaign, not an across-campaign mean; reruns of a selected program characterize execution noise only.

4.2 Results

Open-weight ≤10B regime. Table 1 compares HarnessTTT-9B with every open-weight system of at most 10B parameters that reports these tasks. With the same frozen Qwen3.5-9B executor, HarnessTTT-9B improves on OpenEvolve on all eleven tasks, e.g., from 1.173 to 2.636 on CP26 and from 134.5 to 713.6 on AHC058. It also improves on the discovery-fine-tuned Finch-9B on all eleven, by up to 91.7% on Hadamard and 36.2% on CP26. It holds the best ≤10B value on nine tasks, including Hadamard, where its 0.921 is 60% above the next best (0.576, ThetaEvolve), and a tie at 2.635983 on CP26 with the three executor-updating systems. It trails on two: AC2, where those three systems reach 0.946–0.947 versus 0.934, and Txn, where Finch-4B reaches 4,762 versus 4,255.

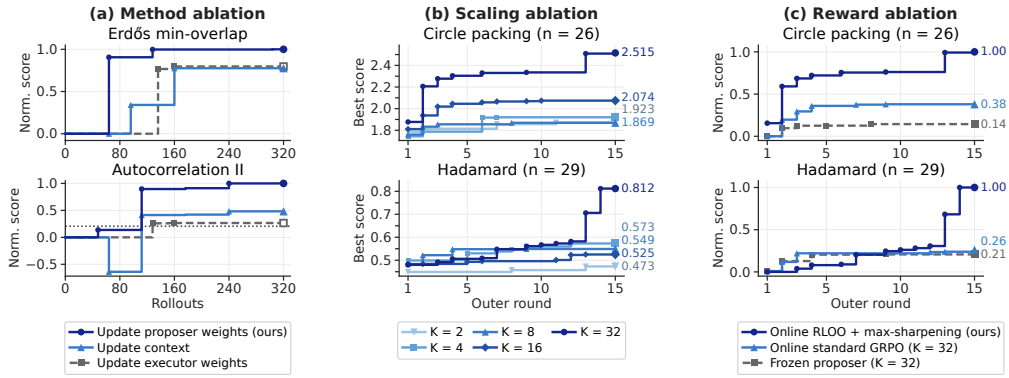


Figure 3: Ablations. **(a)** Reward routing on Erdős and autocorrelation II: normalized score versus rollouts spent; a round costs 8 rollouts for the proposer-weight and context arms and 16 for the executor-weight arm; **(b)** Best score over 15 rounds on circle packing and Hadamard for $K \in \{2, \ldots, 32\}$ proposals per round; round cost grows with K; **(c)** Proposer objective (§3.1), normalized as in (a); the RLOO curve is the $K{=}32$ run in (b).

It also matches or exceeds the best proprietary-executor OpenEvolve result on five of the six tasks where one is reported, all but Txn. Ranks are not compute-matched, since the baselines differ in offline supervision, task metadata, and search budget. Detailed analysis can be found in the appendix.

Adaptation stages. Table 2 isolates what each stage of proposer learning adds. In-context harness editing improves on the fixed initial harness on all thirteen tasks, and training the proposer's weights improves further over in-context adaptation. This highlights the importance of training the proposer's weights rather than relying on in-context adaptation alone.

Proposer model scale. We scale up the model by replacing Qwen3.5-9B with Qwen3.8-27B. Because the 27B model is more expensive to run, we reduce the number of harnesses sampled per round from 32 to 8. Even with this smaller sampling budget, 27B model improves substantially over 9B baseline.

Weak-to-strong transfer. Because the learned harness is explicit and separate from the executor's weights, it can be moved to a stronger executor without retraining. In the spirit of weak-to-strong generalization (Burns et al., 2024), we take the harnesses learned with Qwen3.5-9B and Qwen3.8-27B, replace the executor with GPT-5.5, and continue search (HarnessTTT-Transfer). Table 3 compares both variants with the strongest reported result on each task, regardless of backbone or method. With transfer, HarnessTTT matches or exceeds this frontier on eight of thirteen tasks (six excluding the quarantined LLM-SQL and Txn), is within 1% on four more, and remains behind only on EPLB. It sets new best values on AC2 (0.9636 vs. 0.9629), AHC039 (571,450 vs. 567,503), AHC058, and V2, and ties the frontier on Hadamard and PRISM. Transfer helps most where the small executor falls short: with GPT-5.5, the 9B harness raises AHC058 from 713.55 to 849.68 and Hadamard from 0.921 to 0.936. The transferred harnesses encode search strategies, such as layered construction and filling edge space first (Figure 7), that are not tied to the executor used during optimization, so discovery procedures can be reused across executors without transferring executor weights. Because executor choice and extra search are part of the transfer procedure, these gains do not isolate a pure transfer effect; §4.4 replays a fixed harness on a fixed executor set.

4.3 Ablation Study

Comparison of reward-update locations. A rollout reward can update the executor's weights, the proposer's context, or the proposer's weights. We compare the three routes on two tasks with the same base checkpoint, evaluator, initial harness, seed program, and 20 evaluator calls per executor trajectory. The executor arm is a budget-scaled TTT-Discover-style reference that keeps the initial harness and updates the executor with a three-epoch rank-64 LoRA on $K{=}16$ final-edit trajectories per batch. The proposer arm trains a persistent rank-64 LoRA for three epochs on $K{=}8$ proposals. The context arm makes no gradient update and adds two frozen-model analyst calls after the cold round. Because each harness candidate also costs a proposer rollout, Figure 3a plots every arm against total rollouts (16 per harness round, 8 per executor round; the executor arm's 20 rounds end at 160 and its last value is held). The proposer arm reaches the highest endpoint on both tasks and gets there sooner. By 128 rollouts it has reached 1.00 on Erdős and 0.90 on autocorrelation II (normalized), while the context arm is at 0.34

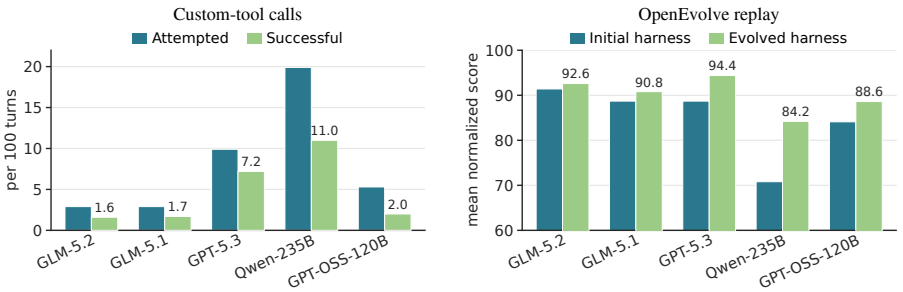


Figure 4: Cross-executor replay of the 9B harness. *Left*: custom-tool calls per 100 turns (eleven tasks, three decode seeds). *Right*: mean normalized score after 48 OpenEvolve iterations from the initial versus the evolved harness (one search seed). Point estimates; the aggregate includes the two quarantined tasks.

and 0.41, and the executor arm is still below the plotted range on Erdős and at 0.26 on autocorrelation II. Updating the proposer thus turns rollout rewards into better search procedures early in the run.

Test-time scaling. We study how discovery performance changes with search width and depth by varying the number of harness proposals per round, $K \in \{2, 4, 8, 16, 32\}$, and running each configuration for 15 outer rounds on CP26 and Hadamard (Figure 3b). The widest configuration, $K = 32$, achieves the highest final score on both tasks: 2.515 on CP26 and 0.812 on Hadamard, compared with 2.074 and 0.573 for the best narrower configurations, respectively. It also makes substantial late-stage improvements on both tasks, while the narrower configurations largely plateau.

Reward design. Figure 3c varies only the proposer objective, with the same frozen executor, initial harness, seed program, budget, and K=32 over 15 rounds; scores are normalized as in Figure 3a. Updating the proposer helps under either objective, and ours helps most: with the RLOO endpoint at 1.00, GRPO reaches 0.38 on CP26 and 0.26 on Hadamard, and the frozen proposer 0.14 and 0.21. GRPO's curve is nearly flat after round 4 on both tasks, whereas RLOO keeps improving. This is consistent with §3.1: without division by the group standard deviation, near-tied groups no longer turn evaluator noise into large advantages, and max-sharpening concentrates credit on the candidate the round keeps. Each condition is a single campaign.

4.4 Cross-Executor Replay of Learned Harnesses

Because the learned harness is explicit, it can be replayed on executors it was never optimized with. We fix the harnesses learned with Qwen3.5-9B (the *evolved harness*) and swap the executor for GLM-5.2, GLM-5.1 (GLM-5 Team, 2026), GPT-5.3 (OpenAI, 2026), Qwen-235B (Yang et al., 2025), or GPT-OSS-120B (OpenAI, 2025), without using any of their outcomes to train the proposer. For each executor, OpenEvolve (Sharma, 2025) runs 48 iterations from the initial or the evolved harness under identical settings, and NexAU (Nex-AGI Team, 2026) records tool use over three decode seeds. The evolved harness raises the eleven-task mean normalized score on all five executors by $+1.2$ to $+13.4$ points ($+5.4$ on average; Figure 4). The gains track custom-tool use: Qwen-235B calls the generated tools most often (19.9 attempts per 100 turns) and gains the most ($+13.4$), followed by GPT-5.3 (9.9; $+5.7$). The harness thus carries a reusable search procedure that other executors can execute, without proposer retraining or executor-weight updates; with one search seed per executor, these gains are descriptive point estimates.

5 Conclusion

We introduced HarnessTTT, which casts test-time discovery as learning the harness rather than the model: a proposer is trained online to write executable harnesses for a frozen executor. With a frozen 9B executor, HarnessTTT improves on OpenEvolve and Finch-9B on all eleven benchmark tasks and holds the best open-weight ($\leq$10B) value on nine; a 27B executor improves further on eleven of thirteen tasks; and with transfer to larger executors it matches or exceeds the reported frontier on eight of thirteen tasks and comes within 1% on four more. Ablations show that, at matched rollouts, updating the proposer's weights reaches better results earlier than updating the executor or the context, and that RLOO with max-sharpening outperforms GRPO and a frozen proposer. Together, these results show that harness search can itself be learned at test time, complementary to scaling the model that executes it.

AI USE STATEMENT

The object of study in this work is LLM agents. Frozen LLMs are the evaluated executors; a trained LLM serves as the proposer that writes harnesses, and the same frozen model repairs generated tools that fail validation. In the cross-executor study, additional frozen models, including commercial API models, run the learned harnesses. Beyond these roles, we used generative AI tools only to edit the manuscript text, prepare figure and table layouts, and format LaTeX. We did not use them to generate experimental data, design experiments, or formulate hypotheses or scientific claims. The authors designed and implemented the system, ran all experiments and analyses, and verified the reported results. The authors take full responsibility for all claims, code, and text.

ETHICS STATEMENT

This work improves frozen agents on optimization tasks with automatic evaluators, all drawn from public benchmarks; it involves no human subjects or personal data. Because the proposer writes code that runs inside the evaluation loop, the main risks are unsafe code execution and specification gaming. We mitigate them by restricting generated tools to a sandboxed interface without network, filesystem, or subprocess access, and by keeping the evaluator, ground truth, and reference scores out of every harness, prompt, and tool. The released artifact does not yet include the trusted runtime needed to verify these controls independently. We disclose evaluator changes and quarantined tasks, and we exclude affected results from our claims.

REFERENCES

Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, et al. GEPA: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025.

Arash Ahmadian, Chris Cremer, Matthias Gallé, et al. Back to basics: Revisiting REINFORCE-style optimization for learning from human feedback in LLMs. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2024.

Ekin Akyürek, Mehul Damani, Adam Zweiger, et al. The surprising effectiveness of test-time training for few-shot learning. In *International Conference on Machine Learning (ICML)*, 2025.

AtCoder Inc. Atcoder heuristic contest. https://atcoder.jp/contests/archive?ratedType=4, 2025.

Bowen Baker, Joost Huizinga, Leo Gao, et al. Monitoring reasoning models for misbehavior and the risks of promoting obfuscation. *arXiv preprint arXiv:2503.11926*, 2025.

Ananth Balashankar, Ziteng Sun, Jonathan Berant, et al. InfAlign: Inference-aware language model alignment. In *International Conference on Machine Learning (ICML)*, 2025.

Collin Burns, Pavel Izmailov, Jan Hendrik Kirchner, Bowen Baker, Leo Gao, Leopold Aschenbrenner, Yining Chen, Adrien Ecoffet, Manas Joglekar, Jan Leike, Ilya Sutskever, and Jeff Wu. Weak-to-strong generalization: Eliciting strong capabilities with weak supervision. In *International Conference on Machine Learning (ICML)*, 2024.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. In *International Conference on Learning Representations (ICLR)*, 2024.

Audrey Cheng, Shu Liu, Melissa Pan, et al. Barbarians at the gate: How AI is upending systems research. *arXiv preprint arXiv:2510.06189*, 2025.

Ching-An Cheng, Allen Nie, and Adith Swaminathan. Trace is the next AutoDiff: Generative optimization with rich feedback, execution traces, and LLMs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

DeepSeek-AI. DeepSeek-V3 technical report. *arXiv preprint arXiv:2412.19437*, 2024. URL https://arxiv.org/abs/2412.19437.

Carson Denison, Monte MacDiarmid, Fazl Barez, et al. Sycophancy to subterfuge: Investigating reward-tampering in large language models. *arXiv preprint arXiv:2406.10162*, 2024.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. In *International Conference on Machine Learning (ICML)*, 2024.

Huan-ang Gao, Jiayi Geng, Wenyue Hua, et al. A survey of self-evolving agents: What, when, how, and where to evolve on the path to artificial super intelligence. *arXiv preprint arXiv:2507.21046*, 2025.

Bogdan Georgiev, Javier Gómez-Serrano, Terence Tao, and Adam Zsolt Wagner. Mathematical exploration and discovery at scale. *arXiv preprint arXiv:2511.02864*, 2025.

GLM-5 Team. GLM-5: From vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026. URL https://arxiv.org/abs/2602.15763.

Qingyan Guo, Rui Wang, Junliang Guo, et al. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. In *International Conference on Learning Representations (ICLR)*, 2024.

Prannay Hebbar, Yogendra Manawat, Samuel Verboomen, Alesia Ivanova, Selvam Palanimalai, Kunal Bhatia, and Vignesh Baskaran. SIA: Self improving AI with harness & weight updates, 2026. URL https://arxiv.org/abs/2605.27276.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. RULER: What's the real context size of your long-context language models? In *Conference on Language Modeling (COLM)*, 2024.

Edward J. Hu, Yelong Shen, Phillip Wallis, et al. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations (ICLR)*, 2022.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *International Conference on Learning Representations (ICLR)*, 2025.

Hyra Team. Hyra: Hunyuan research agent. https://hy.tencent.com/research/hyra, 2026.

Yuki Imajuku, Kohki Horie, Yoichi Iwata, Kensho Aoki, Naohiro Takahashi, and Takuya Akiba. ALE-Bench: A benchmark for long-horizon objective-driven algorithm engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

Sayash Kapoor and Arvind Narayanan. Leakage and the reproducibility crisis in machine-learning-based science. *Patterns*, 4(9):100804, 2023.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, et al. DSPy: Compiling declarative language model calls into self-improving pipelines. In *International Conference on Learning Representations (ICLR)*, 2024.

Wouter Kool, Herke van Hoof, and Max Welling. Buy 4 REINFORCE samples, get a baseline for free! In *ICLR Workshop on Deep RL Meets Structured Prediction*, 2019.

Victoria Krakovna, Jonathan Uesato, Vladimir Mikulik, et al. Specification gaming: the flip side of AI ingenuity. DeepMind Blog, 2020.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, et al. Efficient memory management for large language model serving with PagedAttention. In *ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP)*, 2023.

Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. ShinkaEvolve: Towards open-ended and sample-efficient program evolution. *arXiv preprint arXiv:2509.19349*, 2025.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-Harness: End-to-end optimization of model harnesses, 2026a. URL https://arxiv.org/abs/2603.28052.

Young-Jun Lee, Seungone Kim, Minki Kang, Alistair Cheong Liang Chuen, Zerui Chen, Seungho Han, Taehee Jung, and Dongyeop Kang. Evolution fine-tuning: Learning to discover across 371 optimization tasks. *arXiv preprint arXiv:2606.29082*, 2026b.

Fei Liu, Xialiang Tong, Mingxuan Yuan, et al. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In *International Conference on Machine Learning (ICML)*, 2024a.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024b.

Shu Liu, Shubham Agarwal, Monishwaran Maheswaran, Mert Cemri, Zhifei Li, et al. EvoX: Meta-evolution for automated discovery. *arXiv preprint arXiv:2602.23413*, 2026.

Zichen Liu, Changyu Chen, Wenjun Li, et al. Understanding R1-Zero-like training: A critical perspective. *arXiv preprint arXiv:2503.20783*, 2025.

Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. The AI Scientist: Towards fully automated open-ended scientific discovery. *arXiv preprint arXiv:2408.06292*, 2024.

Haochen Luo, Yi Huang, Sichun Luo, Fengyuan Liu, Lei Li, Zefa Hu, Junlan Feng, and Qi Liu. Harness-aware self-evolving: Co-evolving model weights, harness, and task solutions, 2026. URL https://arxiv.org/abs/2607.03935.

Yecheng Jason Ma, William Liang, Guanzhi Wang, et al. Eureka: Human-level reward design via coding large language models. In *International Conference on Learning Representations (ICLR)*, 2024.

Aman Madaan, Niket Tandon, Prakhar Gupta, et al. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

METR. Recent frontier models are reward hacking. https://metr.org/blog/2025-06-05-recent-reward-hacking/, 2025.

Niklas Muennighoff, Zitong Yang, Weijia Shi, et al. s1: Simple test-time scaling. *arXiv preprint arXiv:2501.19393*, 2025.

Nex-AGI Team. NexAU: A general-purpose agent framework. https://github.com/nex-agi/NexAU, 2026.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, et al. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

OpenAI. gpt-oss-120b & gpt-oss-20b model card, 2025. URL https://arxiv.org/abs/2508.10925.

OpenAI. Introducing GPT-5.3-Codex. https://openai.com/index/introducing-gpt-5-3-codex/, February 2026. Accessed: 2026-08-14.

Krista Opsahl-Ong, Michael J. Ryan, Josh Purtell, et al. Optimizing instructions and demonstrations for multi-stage language model programs. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2024.

Cheng Qian, Chi Han, Yi Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. CREATOR: Tool creation for disentangling abstract and concrete reasoning of large language models. In *Findings of the Association for Computational Linguistics: EMNLP*, 2023.

Jiahao Qiu, Xuan Qi, Tongcheng Zhang, et al. Alita: Generalist agent enabling scalable agentic reasoning with minimal predefinition and maximal self-evolution. *arXiv preprint arXiv:2505.20286*, 2025.

Ao Qu, Han Zheng, Zijian Zhou, Yihao Yan, Yihong Tang, et al. CORAL: Towards autonomous multi-agent evolution for open-ended discovery. *arXiv preprint arXiv:2604.01658*, 2026.

Qwen Team. Qwen3.5: Towards native multimodal agents. Model cards for https://huggingface.co/Qwen/Qwen3.5-9B and https://huggingface.co/Qwen/Qwen3.5-27B, February 2026a. Accessed: 2026-08-30.

Qwen Team. Qwen3.8-Max: A new bar for coding and cowork, August 2026b. URL https://qwen.ai/blog?id=qwen3.8.

Maxime Robeyns, Martin Szummer, and Laurence Aitchison. A self-improving coding agent. *arXiv preprint arXiv:2504.15228*, 2025.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, et al. Mathematical discoveries from program search with large language models. *Nature*, 625:468–475, 2024.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Pier Giuseppe Sessa, Robert Dadashi, Léonard Hussenot, et al. BOND: Aligning LLMs with best-of-N distillation. In *International Conference on Learning Representations (ICLR)*, 2025.

Yu Shang, Yu Li, Keyu Zhao, et al. AgentSquare: Automatic LLM agent search in modular design space. In *International Conference on Learning Representations (ICLR)*, 2025.

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, 2026. URL https://arxiv.org/abs/2608.02276.

Zhihong Shao, Peiyi Wang, Qihao Zhu, et al. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Asankhaya Sharma. OpenEvolve: An open-source implementation of AlphaEvolve. https://github.com/codelion/openevolve, 2025.

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

Parshin Shojaee, Kazem Meidani, Shashank Gupta, et al. LLM-SR: Scientific equation discovery via programming with large language models. In *International Conference on Learning Representations (ICLR)*, 2025.

Joar Skalse, Nikolaus Howe, Dmitrii Krasheninnikov, and David Krueger. Defining and characterizing reward gaming. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

Mirac Suzgun and Adam Tauman Kalai. Meta-prompting: Enhancing language models with task-agnostic scaffolding. *arXiv preprint arXiv:2401.12954*, 2024.

Temoor Tanveer. Levi: Stronger search architectures can substitute for larger llms in evolutionary search. *arXiv preprint arXiv:2605.09764*, 2026.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, et al. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, 2024a.

Yiping Wang, Shao-Rong Su, Zhiyuan Zeng, Eva Xu, Liliang Ren, et al. ThetaEvolve: Test-time learning on open problems. *arXiv preprint arXiv:2511.23473*, 2025.

Zhiruo Wang, Daniel Fried, and Graham Neubig. TroVE: Inducing verifiable and efficient toolboxes for solving programmatic tasks. In *International Conference on Machine Learning (ICML)*, 2024b.

Lilian Weng. Harness engineering for self-improvement. *lilianweng.github.io*, July 2026. URL https://lilianweng.github.io/posts/2026-07-04-harness/.

Amy Xin, Jiening Siow, Junjie Wang, Zijun Yao, Fanjin Zhang, Jian Song, Lei Hou, and Juanzi Li. Eurekagent: Agent environment engineering is all you need for autonomous scientific discovery. *arXiv preprint arXiv:2606.13662*, 2026.

An Yang, Anfeng Li, Baosong Yang, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, et al. Large language models as optimizers. In *International Conference on Learning Representations (ICLR)*, 2024a.

John Yang, Carlos E. Jimenez, Alexander Wettig, et al. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024b.

Haotian Ye et al. Evaluation-driven scaling for scientific discovery. *arXiv preprint arXiv:2604.19341*, 2026. Version 1.

Xunjian Yin, Xinyi Wang, Liangming Pan, et al. Gödel agent: A self-referential agent framework for recursively self-improvement. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2025.

Qiying Yu, Zheng Zhang, Ruofei Zhu, et al. DAPO: An open-source LLM reinforcement learning system at scale. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

Lifan Yuan, Yangyi Chen, Xingyao Wang, et al. CRAFT: Customizing LLMs by creating and retrieving from specialized toolsets. In *International Conference on Learning Representations (ICLR)*, 2024.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, et al. Optimizing generative AI by backpropagating language model feedback. *Nature*, 639, 2025.

Mert Yuksekgonul, Daniel Koceja, Xinhao Li, Federico Bianchi, Jed McCaleb, et al. Learning to discover at test time. *arXiv preprint arXiv:2601.16175*, 2026.

Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-taught optimizer (STOP): Recursively self-improving code generation. In *Conference on Language Modeling (COLM)*, 2024.

Guibin Zhang, Leo Lu, Fangzhou Xie, Kang Zhu, Junhao Wang, Zhifei Xie, Zhaochen Yu, Zihang Liu, Zhongxiang Sun, Qiankun Li, Yue Liao, Heng Chang, Xiaobin Hu, Qibing Ren, Wangchunshu Zhou, and Shuicheng Yan. JIT-Agent: Scaling harness intelligence via just-in-time harness evolution, 2026a. URL https://arxiv.org/abs/2608.25593.

Hangfan Zhang, Shao Zhang, Kangcong Li, Chen Zhang, Yang Chen, Yiqun Zhang, Lei Bai, and Shuyue Hu. Self-Harness: Harnesses that improve themselves, 2026b. URL https://arxiv.org/abs/2606.09498.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Gödel machine: Open-ended evolution of self-improving agents. *arXiv preprint arXiv:2505.22954*, 2025a.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, et al. AFlow: Automating agentic workflow generation. In *International Conference on Learning Representations (ICLR)*, 2025b.

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, pages 86069–86100, 2026c.

Ziqian Zhong, Aditi Raghunathan, and Nicholas Carlini. ImpossibleBench: Measuring LLMs' propensity of exploiting test cases. In *International Conference on Learning Representations (ICLR)*, 2026.

Wangchunshu Zhou, Yixin Ou, Shengwei Ding, et al. Symbolic learning enables self-evolving agents. *arXiv preprint arXiv:2406.18532*, 2024.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, et al. Large language models are human-level prompt engineers. In *International Conference on Learning Representations (ICLR)*, 2023.

Yuxin Zuo, Kaiyan Zhang, Li Sheng, et al. TTRL: Test-time reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

A Overview

This appendix provides additional details on HarnessTTT. It is organized as follows.

1. **Appendix B: HarnessTTT on One Page.** A one-page summary of the two test-time evolution loops, the properties a learned harness provides, the tasks on which it reaches the reported frontier, the tools the proposer wrote, and one circle-packing trajectory.
2. **Appendix C: The Harness Spec Language.** The fields a proposer may edit (system prompt, skills, tools, middleware, and control parameters), their types and limits, and how a partial specification is merged over the incumbent and hashed to remove duplicates.
3. **Appendix D: Capability Surface of Generated Tools.** The `ctx` interface available to generated tools, the operations it withholds, and the static checks and sandboxed self-tests a tool must pass before it is admitted.
4. **Appendix E: Proposer Training Details.** The round configuration (outer rounds, harnesses per round, and evaluator budget per rollout), parent selection and advantage grouping, the RLOO and max-sharpening hyperparameters, and the LoRA, optimizer, and KL settings of the proposer update.
5. **Appendix F: Published Reference Scores.** The reference score u_τ used to normalize the reward on each task, its source, and its optimization direction; these scores enter only the reward and are never shown to the executor.
6. **Appendix G: Discovered Harnesses.** Examples of harnesses found by the trained proposer, including the generated tools, skills, and middleware, and the search strategies they encode.
7. **Appendix H: Audited Reward-Routing Case Study.** Reward-routing trajectories on all four tasks against rollouts spent, and an artifact-level audit of the events behind them.

B HarnessTTT on one page

Figure 5 summarizes HarnessTTT on a single page. Panel (a) contrasts two test-time evolution loops that differ only in what the reward updates. In executor RL, as in TTT-Discover (Yuksekgonul et al., 2026), the executor θ samples N rollouts, each covering one strategy; the reward updates θ, and the final weights $\theta^\star$ cannot be transferred. In HarnessTTT, the proposer ϕ writes K harnesses—named, readable files for the prompt, skills, tools, and middleware—that the frozen executor M_0 rolls out. A single harness-guided rollout can move through several strategies (A→B→C). Candidates are scored and gated, the reward updates only ϕ, and the best harness $H^\star$ is a transferable package. Panel (b) lists the four properties that follow. Exploration happens within a rollout. Every edit is traceable as a named file. Generated code is kept safe by schema, scope, and sandbox checks. The harness transfers to other frozen executors.

Panel (c) marks the tasks on which the learned harness, with or without transfer, matches or exceeds the strongest reported system of any backbone (Table 3). It exceeds the frontier on six tasks: AC2 against Hyra (Hyra Team, 2026); AHC039, AHC058, and V2 against SimpleTES (Ye et al., 2026); LLM-SQL against LEVI (Tanveer, 2026); and Txn against CORAL (Qu et al., 2026). It ties the frontier on Hadamard (SimpleTES) and on PRISM (OpenEvolve (Sharma, 2025), ShinkaEvolve (Lange et al., 2025), CORAL, and LEVI). LLM-SQL and Txn are quarantined. On four more tasks, it is within 1% of the frontier: Erdős and AC1 (Hyra), circle packing (CORAL; EurekAgent (Xin et al., 2026)), and scRNA (SimpleTES). It remains behind only on EPLB (LEVI).

Panel (d) shows the task-specific harnesses the proposer writes on four tasks, with the executor frozen. Each card gives the goal, the visible effect on the solution, and the generated tool and skill or middleware: a staggered-packing constructor and packing guide for circle packing, a basin-and-bottleneck analyzer with a multi-start strategy for Hadamard, a column-ranking tool and row-sorting playbook for LLM-SQL, and a conflict-graph builder with graph-first middleware for transaction scheduling. Panel (e) follows one circle-packing (n=26) campaign with a 9B proposer and executor. Named harness edits raise the sum of radii from 0.960 at the seed to 1.467 in round 1 (broaden the search), 2.235 in round 5 (test four layouts), 2.278 in round 7 (escape a plateau), and 2.541 in round 10 (import a warm start). Replayed unchanged on a frozen GLM-5.2 executor, the same harness reaches 2.635984 in all nine runs, within 0.001% of the reported frontier of 2.635999 (Xin et al., 2026).

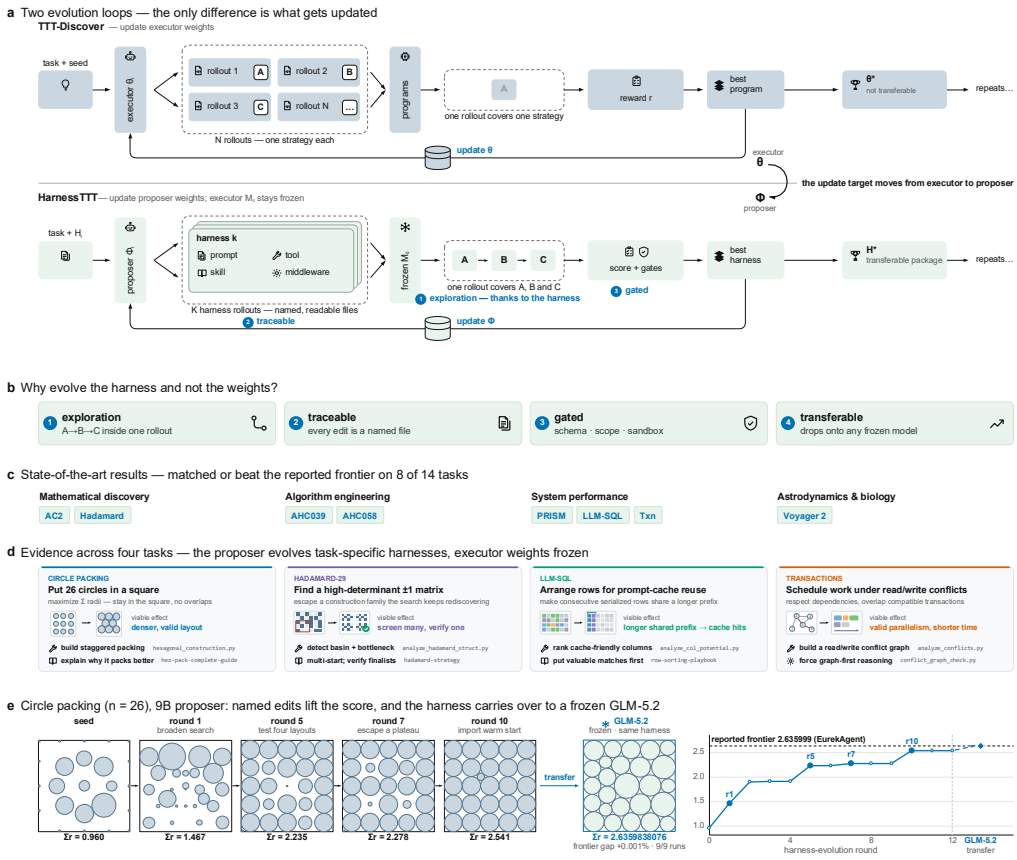


Figure 5: HARNESSTTT on one page. (a) Executor RL versus HARNESSTTT: the same loop, but the reward updates the executor or the proposer. **(b)** Properties of a learned harness. **(c)** Tasks at or above the reported frontier. **(d)** Task-specific harnesses written by the proposer. **(e)** A circle-packing campaign and its transfer to a frozen GLM-5.2 executor.

C THE HARNESS SPEC LANGUAGE

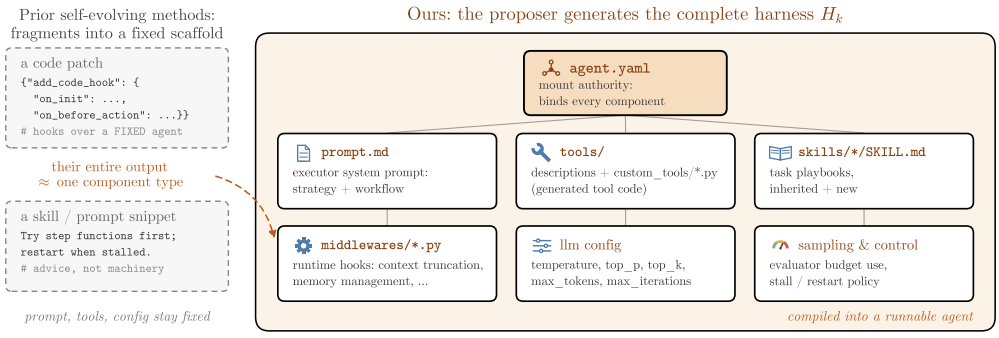


Figure 6: What a generated harness contains. Prior methods emit fragments—a patch or a skill snippet (Shao et al., 2026; Wang et al., 2024a)—into a fixed scaffold. Our proposer can change every component of the executable harness, which is bound by `agent.yaml` and compiled into a runnable agent.

The proposer's output is a *harness spec*: a declarative document that fixes every component the frozen executor sees (Figure 6). A candidate spec is a *partial* YAML document merged over its parent spec. Fields the candidate sets override the parent's, and fields it omits are inherited. A proposal

that rewrites only the skill playbook therefore still defines a complete harness, and what a candidate changes can be read directly from the fields it sets. Table 4 lists the fields.

Table 4: Fields of a harness spec. The upper block overrides components the parent already has; the lower block changes the set of components.

Field	Type	Semantics
Overrides of existing components		
`system_prompt`	text	executor system prompt
`skill_description`	text	skill card shown in the executor's tool list
`skill_body`	text	skill playbook body
`tool_descriptions`	map	descriptions of the fixed tools
`sampling`	map	temperature, top-p, top-k, max tokens
`agent`	map	iteration cap
`middleware`	map	budget-reminder onset; stall-restart threshold and cap; output truncation
Changes to the component set		
`new_tools`	list	≤ 3; name, description, JSON schema, Python implementation
`new_skills`	list	≤ 2; additional skill packages
`new_middlewares`	list	≤ 2; name, hook, Python implementation
`remove_tools`	list	tools to remove; only optional tools qualify

Overriding existing components. The upper block rewrites what the executor reads and how it is run. Three text fields carry the natural-language part of the harness. The system prompt sets the executor's workflow. The skill package is split into a short card that advertises the skill in the executor's tool list and a playbook body that holds the procedure itself. `tool_descriptions` rewrites how the fixed tools are described without touching their code, so the proposer can steer when and how the executor calls them. `sampling` and `agent` set the decoding parameters and the iteration cap. `middleware` tunes the built-in controls: when the budget reminder starts, when a stalled run is restarted and how many restarts are allowed, and the output-truncation limit.

Changing the component set. The lower block adds or removes components. A candidate may add up to three tools, each with a name, a description, a JSON schema for its arguments, and a Python implementation. It may also add up to two skill packages, and up to two middlewares, each attached to a named hook with its own Python implementation. Tools can be removed only if they are marked optional. These caps bound how much new code a single candidate can introduce.

Constraints on generated code. Generated tool names must match `^[a-z][a-z0-9_]{2,31}$`: they start with a lowercase letter, contain 3–32 lowercase letters, digits, or underscores, and may not shadow a reserved name. Each generated implementation is limited to 6,000 characters. A generated tool exposes the single `run(ctx, args)` entry point and reaches the task only through `ctx` (Appendix D). Before it is mounted, it must pass the static gates and sandboxed self-test of §3.2. A tool that still fails after repair is dropped rather than shipped.

Validation, compilation, and hashing. Validation is fail-closed: anything that does not pass is excluded rather than let through. The proposer calls `validate_harness`, which checks component mounts, field schemas, and generated code, and finishes with `submit_harness` (§3). A deterministic materializer then merges the candidate over its parent, compiles the result into a runnable package bound by `agent.yaml`, and writes a canonical spec of the effective harness. Within one materializer revision, identical effective specs yield identical canonical-spec bytes and therefore the same hash. Provenance metadata stored with the package may still differ, so the hash identifies the effective harness rather than any particular package. Because the hash is taken after merging, two candidates in a group that reach the same effective harness through different partial documents share a hash. The hash is used to de-duplicate them.

D Capability surface of generated tools

A generated tool touches the task only through the `ctx` object passed to its `run(ctx, args)` entry point (Table 5). Through this interface a tool can read the current and best programs, stage edits, score them, and read the task's inputs. It cannot see how the scores are computed.

Table 5: The `ctx` interface available to generated tools.

Group	Method	Behavior
Program state	`get_program()`	current program
	`get_best_program()`	best program so far
	`best_score()`	score of the best program
Editing	`stage_edit(code)`	stage an edit to the program
Scoring	`evaluate()`	full evaluation; uses one of the B evaluator calls
	`probe()`	subsampled evaluation, $\sim 8\times$ cheaper; outside B but separately capped
	`budget_left()`	remaining evaluator calls
Task inputs	`list_task_inputs()`	names of the task inputs
	`read_input_sample(name, n)`	n records of a named input
	`read_input_df(name, n)`	the same, as a `pandas` DataFrame
Utilities	`scratch_write/read`	scratch storage
	`log`	log messages

Scoring and budget. The two scoring calls differ in cost and accounting. `evaluate()` runs the full evaluation and uses one of the B evaluator calls. `probe()` scores the program on a subsample at roughly one-eighth the cost. It does not count toward B, but it has its own cap. A tool can therefore screen edits cheaply and save budgeted calls for promising ones, but the cap stops it from using probes as unlimited free evaluations.

Isolation. Path resolution refuses evaluator and ground-truth locations. The configured sandbox gives generated tools no network, general-filesystem, or subprocess access.

Static gate and self-test. The static gate admits imports only from `math`, `re`, `json`, `itertools`, `functools`, `collections`, `heapq`, `bisect`, `random`, `statistics`, `string`, `typing`, `dataclasses`, `numpy`, and `pandas`. It also applies the forbidden-access checks of §3.2. A tool that passes the gate then runs a self-test against a mock context under `RLIMIT_CPU` 20 s and `RLIMIT_AS` 6 GB.

Verifiability. The public artifact does not currently include the trusted runtime, sandbox, gate, scrubber, localization guard, or served-model request logs. These settings therefore describe the evaluated protocol and cannot yet be verified independently.

E PROPOSER TRAINING DETAILS

Table 6 gives the configuration of both proposer-training campaigns: the Qwen3.5-9B campaigns of §4.2, and the Qwen3.8-27B campaign whose harnesses Appendix G lists. The two campaigns share the search, reward, and update recipe. They differ in model scale, proposals per round, rollout budget, the baseline of the reward, and the handling of all-tied groups. The paragraphs below explain how the pieces fit together.

Round structure. Each task runs for 20 outer rounds. In each round the proposer samples K harnesses, and each candidate harness is rolled out once by the frozen executor. In the 27B campaign, each candidate is additionally scored against a paired control: its parent harness is rolled out once more at the same decode seed (see *Rewards and advantages* below). A round therefore launches $3K = 24$ agent trajectories in the 27B campaign (one proposer episode, one candidate rollout, and one parent-control rollout per candidate) and $3K = 96$ in the 9B campaigns, which compare against the shared incumbent score and need no control rollout. To limit the cost of the larger executor, the 27B campaign samples $K = 8$ proposals rather than 32. It also gives each rollout $B = 5$ evaluator calls of up to 420 s, where the 9B campaigns give 20 calls of up to 600 s. Every rollout is capped at 3 h of wall-clock time. A round is accepted only if at least six of the eight proposals validate (the `min6` gate). The same frozen model may repair a generated tool that fails validation at most twice.

Parent selection and grouping. Programs and harnesses are promoted separately. The program registry is a strict single-program ratchet: a rollout's program replaces the incumbent only if it strictly improves the score. In the 27B campaign, the incumbent *harness* is promoted only when the round's

Table 6: Proposer-training hyperparameters. Values centered across both columns are shared; —: not stated for the 9B campaigns.

Parameter	27B campaign	9B campaigns
Search		
Outer rounds per task	20	
Proposals per round K	8	32
Trajectories per round	$3K = 24$	$3K = 96$
Parent beam	incumbent + best candidate within 0.02 that adds a new component	
Advantage group size K_g	K in round 1, $K/2$ afterwards	
Round acceptance	≥ 6 of 8 proposals validate	
Program promotion	strict score improvement	
Executor rollout		
Executor (frozen)	Qwen3.8-27B	Qwen3.5-9B
Serving	vLLM, TP 2 $\times$ 4 replicas, one 8-GPU node	—
Context window	49,152 tokens	—
Evaluator calls B	5	20
Time per evaluation	420 s	600 s
Wall-clock limit	3 h	
Proposer episode		
Proposer	Qwen3.8-27B + LoRA	Qwen3.5-9B + LoRA
Harness	fixed two-tool H1, workspace preloaded	
Agent iterations	≤ 12	
Sampling	temperature 1.0, top-p 0.95	
Max generated tokens	60,000	
Tool repair	≤ 2 iterations by the same frozen model	
Reward and advantage		
Normalized gain g_k	paired: $(s_k - c_k)/(u_\tau - c_k)$	unpaired: $(s_k - b_\tau)/(u_\tau - b_\tau)$
Shaped reward r_k	$3\tanh(g_k/3)$ if $g_k > 0$, else $\max(-1, g_k)$	
Tie threshold δ	10^{-3}	
Sharpening weight α	0.3	
Softmax temperature T	$\max(\mathrm{range}(r)/4,\ 10^{-6})$	
All-tied groups	excluded ($A_k = 0$)	frontier rescue, $\lambda = 0.3$
Proposer update		
Framework	slime, FSDP backend, 8 GPUs	
LoRA	rank 64, $\alpha_{\mathrm{LoRA}} = 128$; `q,k,v,o,gate,up,down`	
Optimizer	Adam, $\beta_1 = 0.9$, $\beta_2 = 0.98$, weight decay 0	
Learning rate	10^{-5}, constant	
Batch	micro 1; global = all trainable episodes of the round ($\leq K$)	
Epochs per round	3	
KL penalty	0.1 (`low_var_kl`) to the frozen base	
Episode length	$\leq 18{,}000$ tokens, terminal-preserving	

best candidate both beats its same-seed paired control and exceeds the historical best; displaced parents are archived. From round 2 on, the K proposal slots are dealt round-robin to a beam of two parents: the promoted or retained incumbent, and the best candidate within 0.02 of it that adds a new component. An advantage group contains the proposals that share one sampling condition (parent spec, prompt, and budget hash). Hence $K_g = K$ in round 1 and $K/2$ afterwards, which is 8 and then 4 for the 27B campaign, and 32 and then 16 for the 9B campaigns.

Rewards and advantages. The two campaigns differ in the normalized-gain baseline. The 9B campaigns compare each candidate with the shared incumbent score b_τ. The 27B campaign compares it with its own paired control: a rollout of the parent harness at the same decode seed, with score c_k. For the 27B campaign,

$$g_k = \frac{s_k - c_k}{u_\tau - c_k}, \qquad r_k = \begin{cases} 3\tanh(g_k/3) & g_k > 0, \\ \max(-1,\ g_k) & g_k \leq 0, \end{cases}$$

and the 9B gain replaces c_k with b_τ. Here u_τ is the reference score of Appendix F. The positive branch compresses large gains smoothly, and the negative branch keeps regressions linear down to -1. Invalid candidates, candidates repaired by the frozen model, and candidates whose generated component never activated receive $r_k = -1$ (or at most 0), so they cannot earn credit for a gain. Among the creditable candidates, rewards within $\delta = 10^{-3}$ of each other are collapsed to their mean. The advantage then follows Eq. equation 3 with group size K_g: $A_k = 0.7\, A_k^{\mathrm{LOO}} + 0.3\,\big(\mathrm{softmax}(r/T)_k - 1/K_g\big)$ with $T = \max(\mathrm{range}(r)/4,\, 10^{-6})$. A group whose creditable rewards all lie within δ carries no signal. The 27B campaign excludes such a group ($A_k = 0$), while the 9B campaigns rescue it with a historical-frontier term ($\lambda = 0.3$).

Proposer update. Training runs on slime with the FSDP backend on 8 GPUs, for three epochs per round over all trainable episodes of the round. Each episode is converted into a replay of its full multi-turn trajectory, and the loss mask covers assistant turns that are closed by a later user or tool message. Episodes are truncated to 18,000 tokens in a terminal-preserving way: tool observations are clipped first, and then the longest fitting suffix that starts at an assistant action is kept. The rewarded `submit` action therefore always survives truncation. The loss is a clipped policy-gradient objective with the precomputed advantage A_k broadcast over the masked tokens. slime's own reward normalization is disabled, consistent with the absence of variance scaling in Eq. equation 3. A KL penalty of 0.1 (`low_var_kl`) anchors the policy to the frozen base. The adapter is cumulative: each round loads only the weights of the previous round's adapter (`--load-weights-only`). Every round's checkpoint is retained. The merged checkpoint is served as the next round's proposer, while the executor stays at the base checkpoint.

F Published reference scores

The reward of §3.1 divides a candidate's gain over its baseline (the incumbent score b_τ, or the paired control in the 27B campaign; Appendix E) by the baseline's remaining gap to a task reference u_τ. The reference therefore sets the scale of credit on each task; it is not a target that the proposer or the executor ever sees.

In the 27B campaign, u_τ is the largest of four fixed values: the seed score, the best human value, the published $\leq$10B SOTA, and the Finch-9B score. All four are expressed in the evaluator's combined score. Table 7 lists the resulting references, taken from the campaign's `normalization_references.json`. The 9B campaigns use the published $\leq$10B SOTA alone, which changes the reference on only two tasks, Erdős and Hadamard (see the table notes). When u_τ does not exceed the baseline, the remaining gap is not positive. The reward then falls back to the relative gain over $|b_\tau|$, clipped to $[-1, 5]$. The reference enters the reward only. It is never written into a harness, a prompt, or a tool, and a leak guard rejects any proposal or repair whose text contains it.

G Discovered harnesses

This section lists the final incumbent harness that the trained Qwen3.8-27B proposer produced on two of its evolution tasks, circle packing and Erdős minimum overlap (20 rounds each). The final harness is the package that the last round's collector retained as the parent. Table 8 summarizes the two harnesses. For each task we then give the lineage of the final harness and excerpts of the compiled artifacts on disk: the effective spec with long bodies replaced by pointers, the generated tool code, the generated skill playbooks, and the policy sections the proposer wrote into the executor's system prompt.

Both harnesses keep the base configuration: sampling at T=0.7, top-p 0.95, top-k 20, and at most 8,192 tokens; 36 executor iterations; and the base middleware thresholds (budget reminder from 3 evaluations left, stall restart after 8 stalls, at most 2 restarts, tool output clipped at 8,000 characters). The proposer's edits went into the prompt, the skills, and the tools.

Table 7: Reference scores u_τ of the 27B campaign. x is the evaluator's combined score (higher is better); f maps it to the native metric of Table 3.

Task	Metric	$f(x)$	u_τ	$f(u_\tau)$	Source	Backbone
Mathematical discovery						
Erdős ↓	c_5	$0.380922\,/\,x$	0.999987	0.380927	Best human	—
AC1 ↓	c_1	$1.505293\,/\,x$	1.001437	1.5031	ThetaEvolve	R1-Qwen3-8B
AC2 ↑	c_2	$0.896280\,x$	1.056478	0.9469	TTT-Discover	Qwen3-8B
CP26 ↑	sum of radii	$2.635\,x$	1.000373	2.635983	ThetaEvolve	R1-Qwen3-8B
Hadamard ↑	det. ratio	x	0.935673	0.935673	Best human	—
Algorithm engineering						
AHC039 ↑	total score	$225{,}000\,x$	2.519987	566,997	Best human	—
AHC058 (M) ↑	total score	$450\,x$	1.883722	847.67	Best human	—
System performance						
EPLB ↑	balancedness	x	0.1270	0.1270	OpenEvolve	Finch-8B
PRISM ↑	combined	x	24.70	24.70	OpenEvolve	Finch-8B
LLM-SQL ↑	combined	x	0.7341	0.7341	OpenEvolve	Finch-8B
Txn ↑	combined	x	4,761.9	4,761.9	OpenEvolve	Finch-4B

Table 8: The two final 27B harnesses. Best program: the campaign's retained program in native units (Table 7) and the round that produced it.

Task	Harness	Spec hash	Tools	Skills	Best program	Prompt policy written by the proposer
CP26 ↑	r13/c06	`128394fe5159501a`	2	2	2.635983 (r18)	preflight–repair–rank loop; transform as comparator
Erdős ↓	r19/c01	`f940301924f28c6a`	2	5	0.380902 (r19)	five-evaluation funding rule; hard-max-first reporting

G.1 Circle packing (n=26): a verifier and a transform in front of every evaluation

The lineage (Table 9) shows the harness being assembled in the order a human engineer might choose. A feasibility checker comes first (round 0), then a construction playbook that tells the executor how to use it (round 5), then a transform that proposes repaired geometry (round 6), and finally a first-moves policy that stops the executor from discarding the incumbent's optimizer (rounds 8 and 13). The checker, `packing_feasibility` (Listing 3), uses a water-filling relaxation to return the largest radii consistent with the walls and every pair, so the executor can rank layouts and repair the listed breaches without spending an evaluation. Eight further generated tools proposed in rounds 1–18 (`packing_repair`, `packing_variant_generator`, `local_packing_optimizer`, `packing_family_scan`, ...) were rolled out but not promoted.

Table 9: Circle packing: lineage of the final harness. Harness score (combined; native $= 2.635\,x$) after each round, and the generated components of the round's best candidate: tools, skills, and *prompt edits*. Shaded: the round that produced the final harness.

Round	Components of the round's best candidate	Promoted	Harness score
0	`packing_feasibility`	✓	0.964486
1–4	`packing_repair`; `packing_variant_generator` + skill; `local_packing_optimizer`	✗	0.964486
5	`circle-packing-construction`	✓	0.990712
6	`packing_transform`	✓	0.997374
7	*prompt only*	✓	0.998137
8	`dense-constructor-first-moves`	✓	0.999731
9–12	`packing_family_scan`; *prompt edits*	r12 only	0.999731
13	`dense-constructor-first-moves` (updated)	✓	1.000371
14–19	`packing_family_probe`, `packing_refine`, `packing_family_diversifier`, `topology-family-enumeration`	✗	1.000371

Listing 1: Circle packing: effective spec of the final harness. Long bodies are replaced by pointers.

```
schema: h2spec/1.0
skill_description: Iteratively optimize a program's EVOLVE-BLOCK to maximize an automatic
    ↪ evaluator score,
  under a fixed evaluation budget. Use for construction, algorithm-speed, and heuristic
    ↪ discovery tasks
  scored by combined_score (higher is better) through the edit_solution / evaluate_solution /
    ↪  finish tools.
tool_descriptions:
  edit_solution: 'Change the code inside the # EVOLVE-BLOCK region. Two accepted formats: (1)
    ↪  PREFERRED
    - one or more targeted SEARCH/REPLACE diff blocks, each matching the current program
    ↪ exactly: ''<<<<<<<
    SEARCH'' / exact original lines / ''======='' / new lines / ''>>>>>>> REPLACE''. Do not
    ↪ rewrite the
    whole region for a small change. (2) A full rewrite - the complete new EVOLVE-BLOCK body
    ↪  as plain
    code (fenced ```python is allowed; the markers are optional). The fixed entry function
    ↪ and imports
    outside the block are preserved automatically. After editing, call evaluate_solution to
    ↪ score it.'
  evaluate_solution: Run the current program through the task evaluator and report its score.
    ↪  Returns
    combined_score (higher is better), validity, any error, your best score so far, and how
    ↪ many evaluations
    remain. Use the feedback to improve the code, then evaluate again. The best-scoring
    ↪ version is kept
    automatically. Your number of evaluations is limited.
  probe_solution: Cheaply score the CURRENT program on SUBSAMPLED data (first ~2000 rows of
    ↪ any dataset).
    Fast and does NOT consume the evaluation budget, but the score is approximate and NOT
    ↪ comparable to
    full evaluate_solution scores - use probes to rank your own variants when full
    ↪ evaluation is slow,
    then confirm the best one with evaluate_solution.
  finish: End the session. Call this when your evaluation budget is exhausted or you cannot
    ↪ improve the
    best score further. The best-scoring program is submitted automatically.
sampling:
  temperature: 0.7
  top_p: 0.95
  top_k: 20
  max_tokens: 8192
agent:
  max_iterations: 36
middleware:
  budget_reminder_from_left: 3
  stall_after: 8
  max_restarts: 2
  long_tool_output_max_chars: 8000
system_prompt: <9550 chars; see prompt.md>
skill_body: <2388 chars; base discovery-optimization skill>
new_tools:
- name: packing_feasibility
```

```
  description: 'Deterministic geometry checker for a candidate circle packing: given 2D
    ↪ centers (and optional
    radii), returns the maximal feasible radii under wall+pair constraints, a validity flag,
    ↪  the specific
    wall/pair violations with breach magnitudes, the sum of radii (the scored quantity), and
    ↪  a coarse
    radius histogram to detect candidate-family collapse. Call it on the exact centers you
    ↪ intend to submit
    before every evaluate_solution; it does not consume evaluation budget.'
  input_schema:
    type: object
    properties:
      centers:
        type: array
        description: List of [x, y] pairs for every circle in the candidate packing. Each
    ↪ coordinate must
          be a finite number. Use the exact centers you intend to submit.
        items:
          type: array
          items:
            type: number
          minItems: 2
          maxItems: 2
        minItems: 2
        maxItems: 200
      radii:
        type: array
        description: Optional. Explicit per-circle radii, same length as centers. When
    ↪ omitted, the tool
          computes the maximal feasible radii for the given centers.
        items:
          type: number
          minimum: 0
        minItems: 2
        maxItems: 200
    required:
    - centers
    additionalProperties: false
  implementation_py: <6587 chars; see packing_feasibility.py>
- name: packing_transform
  description: Deterministic packing generator/repairer. Given current centers (and optional
    ↪  radii), it
    pushes interior-slack circles toward their nearest wall, separates overlapping pairs
    ↪ along their join
    line, then re-solves the max-radius feasibility LP (wall + pair + r>=0). Returns
    ↪ transformed centers,
    maximal feasible radii, resulting sum_radii, validity, residual violations, and a coarse
    ↪  family_key
    for diversity. It is a TRANSFORM (new geometry), distinct from packing_feasibility which
    ↪  only diagnoses.
    Costs no evaluation budget.
  input_schema:
    type: object
    properties:
      centers:
        type: array
        description: List of [x, y] pairs for every circle in the candidate packing (
    ↪ typically 26). Each
          coordinate must be a finite number in [0,1].
        items:
          type: array
          items:
            type: number
          minItems: 2
          maxItems: 2
        minItems: 2
        maxItems: 200
      radii:
        type: array
        description: Optional. Explicit per-circle radii, same length as centers. When
    ↪ omitted, the tool
          solves for maximal feasible radii on the transformed centers.
        items:
          type: number
          minimum: 0
        minItems: 2
        maxItems: 200
    required:
    - centers
    additionalProperties: false
  implementation_py: <5209 chars; see packing_transform.py>
```

```
new_skills:
- name: circle-packing-construction
  description: 'Geometry playbook for maximizing the sum of 26 circle radii in a unit square:
    ↪  verify every
    candidate with the packing_feasibility tool before any evaluation, repair from its
    ↪ violation list,
    push slack circles toward walls, and switch structural families on histogram/score
    ↪ plateau.'
  body: <4408 chars; see circle-packing-construction.SKILL.md>
- name: dense-constructor-first-moves
  description: 'Circle-packing first-moves playbook for n=26 in a unit square: compare a
    ↪ keep-and-refine
    of the incumbent constructor against at most one alternate dense template, preflight
    ↪ both with the
    feasibility tool, and spend the limited evaluation budget on the higher-verified
    ↪ candidate.'
  body: <2840 chars; see dense-constructor-first-moves.SKILL.md>
```

Listing 2: Circle packing: task policy the proposer wrote into the executor's system prompt (excerpt of `prompt.md`).

```
# Circle-packing task policy (n=26 in a unit square)

Scores the **sum of radii** of 26 circles in a unit square. Hard constraints:
every circle inside the square; no two overlap. A violation => validity 0.

**Preflight before every `evaluate_solution`:** call `packing_feasibility` on
your exact centers; only evaluate when `valid=true` and `sum_radii` exceeds
your best. Costs no evaluation budget.

**No post-hoc clipping** (`np.clip(centers, 0.01, 0.99)`): it silently moves
circles and can create overlaps. Place centers already inside the square.

**Do NOT discard a working optimizer-based constructor** (SLSQP/local
refinement over a seeded lattice + max-radius LP) just to emit a hand-written
literal table; a refined seed usually out-scores a from-scratch layout.
Treat `packing_transform` as a **comparator**, not a mandate: adopt its output
only when its `sum_radii` strictly beats your best -- its wall-push can break
tight triangular contacts and lower the sum. Compare >= 2 construction families
(keep-and-refine vs. one different row/shell/corner template), promote by the
tool's `sum_radii`, confirm the winner, and reserve the last evaluation for
consolidating the highest-`sum_radii` valid candidate.
```

Listing 3: Circle packing: core of the generated tool `packing_feasibility` (round 0).

```python
def _maximal_radii(centers):
    """Largest radii satisfying r_i <= wall_i and r_i + r_j <= d_ij.

    Water-filling relaxation: start at wall distances, then for every pair
    whose constraint is violated, reduce the larger of the two radii to
    restore feasibility. Repeat until stable or the cap is hit.
    """
    n = len(centers)
    radii = [min(c[0], 1.0 - c[0], c[1], 1.0 - c[1]) for c in centers]
    radii = [max(r, 0.0) for r in radii]
    pairs = []
    for i in range(n):
        for j in range(i + 1, n):
            dx = centers[i][0] - centers[j][0]
            dy = centers[i][1] - centers[j][1]
            d = math.sqrt(dx * dx + dy * dy)
            if d > 0.0:
                pairs.append((i, j, d))
    for _ in range(_MAX_ITERS):            # 200 sweeps, ~13k pair checks for n=26
        changed = False
        for i, j, d in pairs:
            s = radii[i] + radii[j]
            if s > d + _EPS:
                if radii[i] >= radii[j]:
                    new_i = max(0.0, d - radii[j])
                    if radii[i] > new_i + _EPS:
                        radii[i] = new_i; changed = True
                else:
                    new_j = max(0.0, d - radii[i])
                    if radii[j] > new_j + _EPS:
                        radii[j] = new_j; changed = True
        if not changed:
            break
```

```
    return radii

def run(ctx, args):
    centers = _parse_centers(args.get("centers"))
    ...
    radii = _parse_radii(args.get("radii"), n)
    if radii is None:
        radii = _maximal_radii(centers); radii_source = "computed_maximal"
    wall, pairs_bad = _violations(centers, radii)
    valid = (len(wall) == 0) and (len(pairs_bad) == 0)
    return {"ok": True, "n_circles": n, "radii_source": radii_source,
            "sum_radii": round(sum(radii), 9), "valid": valid,
            "wall_violations": wall[:20], "pair_violations": pairs_bad,
            "violation_count": len(wall) + len(pairs_bad),
            "radius_histogram": _histogram(radii),
            "hint": ("If valid=false, fix the listed violations: for a pair, move the "
                     "circle with the larger radius away along the line joining the two "
                     "centers; for a wall breach, pull the circle inward. Then re-run "
                     "this tool before calling evaluate_solution."
                     if not valid else
                     "Packing is feasible; sum_radii is the quantity the evaluator rewards.")
    ↪ }
```

Listing 4: Circle packing: generated skills. The mandatory core loop of `circle-packing-construction` (round 5) and the first-moves rule of `dense-constructor-first-moves` (rounds 8 and 13).

```
## Core loop (mandatory order)                    -- circle-packing-construction
1. Propose.  After every edit_solution, extract the exact list of 26 [x, y]
   centers the code will emit.
2. Preflight.  Call packing_feasibility with those exact centers (omit radii so
   it computes maximal feasible radii). This costs no evaluation budget.
3. Read.  valid, sum_radii, wall_violations, pair_violations (sorted by breach
   magnitude), radius_histogram.
4. Repair if invalid.  Fix the largest 1-2 breaches per pass: pair breach ->
   move the larger circle away along the join line; wall breach -> pull inward.
   Stop after 3 repair passes -- a still-invalid layout is structurally weak.
5. Rank without evaluating.  Several center sets, ranked by the tool's
   sum_radii; promote at most the top 1-2 to evaluate_solution.
6. Evaluate only valid, pre-ranked layouts.
Family switch (triggered): two consecutive evaluate_solution results gain
< 0.5%, or the radius_histogram is nearly single-bin while sum_radii is
plateaued -> change structure, not coordinates (rows/shells, asymmetric radii,
corner treatment).

## Core rule                                      -- dense-constructor-first-moves
Keep-and-refine is the default first move. Do NOT rewrite a working
optimizer-based constructor (SLSQP/local refinement over a seeded lattice
followed by a max-radius LP solve) into a hand-written literal center table.
1. Keep the incumbent core; make the first edit a targeted improvement to it.
2. One alternate arm for diversity: at most ONE structurally different template.
3. Preflight both with packing_feasibility; rank by sum_radii.
4. Promote the winner; never evaluate an invalid layout.
Budget: most of the session preflighting and comparing, at most 2-3
evaluations on distinct finalists, the last one reserved for consolidation.
```

G.2 Erdős minimum overlap: a family ranker, a spectral diagnostic, and a funding rule

For its first four rounds, the Erdős harness gained one skill per round: objective and projection, a preservation-first method gate, candidate diversity, and evaluation allocation. Two tools then made these skills executable. `overlap_spectral_diagnostic` (round 4) recomputes the task's linear hard-max overlap $c_5 = \max_k \int h(x)(1 - h(x+k))\,dx$ for one candidate and names the binding shifts. `block_family_sweeper` (round 5; updated in rounds 14, 16, and 19) enumerates a bounded family of $0/1$ block constructions and ranks them by the same objective. Both tools compute the *linear* overlap with a zero-padded FFT (Listing 6), and the proposer's tool description warns the executor against the circular-FFT "illusion" of the seed program. The final round's update of the sweeper coincided with the program that beats the best human construction ($c_5 = 0.380902 < 0.380927$).

Table 10: Erdős: lineage of the final harness, as in Table 9. Native $c_5 = 0.380922/x$.

Round	Components of the round's best candidate	Promoted	Harness score
0	`erdos-min-overlap`	✓	0.988310
1	`erdos-min-overlap-method`	✓	0.999422
2	`erdos-candidate-diversity`	✓	0.999528
3	`c5-eval-allocation`	✓	0.999559
4	`overlap_spectral_diagnostic`	✓	0.999880
5	`block_family_sweeper`	✓	0.999881
6–7	*prompt only*; `relaxed_overlap_sweeper`	✓	0.999894
8	`fast-structural-c5-search`	✓	0.999904
9–13	`worst_shift_repair`; skill removals and re-adds	mixed	0.999904
14	`block_family_sweeper` (updated) + `erdos-min-overlap-method` (re-added)	✓	0.999904
15–16	*prompt only*; `block_family_sweeper` (updated)	✓	0.999925
17–18	`c5-budget-focus` replacing `c5-eval-allocation`	✗	parent dipped to 0.833051 (r18)
19	`block_family_sweeper` (updated)	✓	1.000053

Listing 5: Erdős: effective spec of the final harness.

```
schema: h2spec/1.0
skill_description: Iteratively optimize a program's EVOLVE-BLOCK to maximize an automatic
    ↪ evaluator score,
  under a fixed evaluation budget. Use for construction, algorithm-speed, and heuristic
    ↪ discovery tasks
  scored by combined_score (higher is better) through the edit_solution / evaluate_solution /
    ↪  finish tools.
tool_descriptions:
  edit_solution: 'Change the code inside the # EVOLVE-BLOCK region. Two accepted formats: (1)
    ↪  PREFERRED
    - one or more targeted SEARCH/REPLACE diff blocks, each matching the current program
    ↪ exactly: ''<<<<<<<
    SEARCH'' / exact original lines / ''======='' / new lines / ''>>>>>>> REPLACE''. Do not
    ↪ rewrite the
    whole region for a small change. (2) A full rewrite - the complete new EVOLVE-BLOCK body
    ↪  as plain
    code (fenced ```python is allowed; the markers are optional). The fixed entry function
    ↪ and imports
    outside the block are preserved automatically. After editing, call evaluate_solution to
    ↪ score it.'
  evaluate_solution: Run the current program through the task evaluator and report its score.
    ↪  Returns
    combined_score (higher is better), validity, any error, your best score so far, and how
    ↪ many evaluations
    remain. Use the feedback to improve the code, then evaluate again. The best-scoring
    ↪ version is kept
    automatically. Your number of evaluations is limited.
  probe_solution: Cheaply score the CURRENT program on SUBSAMPLED data (first ~2000 rows of
    ↪ any dataset).
    Fast and does NOT consume the evaluation budget, but the score is approximate and NOT
    ↪ comparable to
    full evaluate_solution scores - use probes to rank your own variants when full
    ↪ evaluation is slow,
    then confirm the best one with evaluate_solution.
  finish: End the session. Call this when your evaluation budget is exhausted or you cannot
    ↪ improve the
    best score further. The best-scoring program is submitted automatically.
sampling:
  temperature: 0.7
  top_p: 0.95
  top_k: 20
  max_tokens: 8192
agent:
  max_iterations: 36
middleware:
  budget_reminder_from_left: 3
  stall_after: 8
  max_restarts: 2
```

```
  long_tool_output_max_chars: 8000
system_prompt: <9410 chars; see prompt.md>
skill_body: <2388 chars; base discovery-optimization skill>
new_tools:
- name: overlap_spectral_diagnostic
  description: Cheaply compute the per-shift overlap profile c(k)=int h(x)(1-h(x+k))dx for a
    ↪  single candidate
    step function h on [0,2], given as either explicit 'blocks' or a value array 'h'.
    ↪ Returns the hard-max
    C5 value (the max-k objective), the dominant (worst) shifts, top-2 shift gap, a soft-max
    ↪  temperature
    recommendation, the integral-of-h feasibility check, and head/tail of the profile. Pure
    ↪ deterministic
    diagnostics to rank candidate geometries/parameterizations before committing a full
    ↪ evaluate_solution.
  input_schema:
    type: object
    properties:
      blocks:
        type: array
        description: Optional list of intervals where h=1; each element is {start,end} (in
    ↪ [0,2]) or a
          2-element [start,end]. Provide this OR 'h'.
        items:
          type: object
      h:
        type: array
        description: Optional array of candidate h values in [0,1] over [0,2]; resampled to
    ↪ num_intervals
          if length differs. Provide this OR 'blocks'.
        items:
          type: number
      num_intervals:
        type: integer
        description: Discretization N for the [0,2] grid (default 600).
        default: 600
      top_k:
        type: integer
        description: Number of worst shifts to report (default 3).
        default: 3
    required: []
    additionalProperties: false
  implementation_py: <3923 chars; see overlap_spectral_diagnostic.py>
- name: block_family_sweeper
  description: Deterministically enumerate a bounded family of distinct 0/1 step-function
    ↪ candidates h
    on [0,2] (bump count, spacing mode, width, symmetry), scale each for exact integral 1,
    ↪ and rank them
    by the TASK's LINEAR (non-circular) hard-max C5 = max_k int h(x)(1-h(x+k))dx over k in
    ↪ [0,2] via zero-padded
    FFT (identical method to overlap_spectral_diagnostic, so the ranking matches the true
    ↪ objective).
    Returns ranked candidates with worst shift, top-2 gap, and profile head/tail. Use it to
    ↪ pick which
    block construction to commit to edit_solution/evaluate_solution.
  input_schema:
    type: object
    properties:
      n_bumps:
        type: integer
        description: Number of bumps in the construction family (1-16).
      spacing:
        type: string
        description: 'Spacing mode: ''periodic'' (evenly spaced), ''clustered'' (around
    ↪ center x=1), or
          ''pair'' (mirror-symmetric pairs).'
      bump_width:
        type: number
        description: Base bump width before integral-1 scaling, in [0.01, 1.0].
      symmetry:
        type: string
        description: Reserved symmetry tag for the family ('none', 'mirror'); does not alter
    ↪  generation
          when spacing is 'pair'.
      widths:
        type: array
        description: Optional list of up to 6 alternative widths to sweep within the family.
        items:
          type: number
      n_intervals:
        type: integer
```

```
        description: Discretization N for the [0,2] grid (3-2400, default 600).
      max_candidates:
        type: integer
        description: Maximum candidates to generate and rank (default 12).
    required: []
    additionalProperties: false
  implementation_py: <8471 chars; see block_family_sweeper.py>
new_skills:
- name: erdos-min-overlap
  description: 'Playbook for reducing the C5 upper bound in the Erdos minimum-overlap
    ↪ problem: minimize
    max_k int h(x)(1-h(x+k)) dx over step functions h:[0,2]->[0,1] with inth=1. Use whenever
    ↪  editing the EVOLVE-BLOCK
    for this task.'
  body: <3635 chars; see erdos-min-overlap.SKILL.md>
- name: erdos-min-overlap-method
  description: 'Method gate for the Erdos minimum-overlap C5 task: accurate incumbent
    ↪ characterization,
    preservation-first edit gate, and a ranked list of high-leverage polishes to test one
    ↪ per evaluation.'
  body: <4809 chars; see erdos-min-overlap-method.SKILL.md>
- name: erdos-candidate-diversity
  description: Playbook for diversifying step-function candidates in the min-overlap C5
    ↪ objective so each
    full evaluation covers several distinct basins instead of re-optimizing one
    ↪ hyperparameter variant.
  body: <3590 chars; see erdos-candidate-diversity.SKILL.md>
- name: c5-eval-allocation
  description: 'Allocation rule for the 5-evaluation Erdos C5 budget: confirm the incumbent,
    ↪  allow one
    micro-polish, then fund each remaining evaluation on a distinct candidate family (value
    ↪ scheme, support
    structure, or optimizer), with a 1e-4 switch rule and hard-max-first reporting.'
  body: <2892 chars; see c5-eval-allocation.SKILL.md>
- name: fast-structural-c5-search
  description: 'Workflow for choosing which block construction to invest expensive
    ↪ evaluations in: rank
    families with block_family_sweeper, diagnose worst shifts with
    ↪ overlap_spectral_diagnostic, make one
    structural edit, confirm once.'
  body: <2265 chars; see fast-structural-c5-search.SKILL.md>
```

Listing 6: Erdős: objective evaluation shared by both generated tools (from `block_family_sweeper`).

```python
def _c5_linear_from_blocks(blocks, n_intervals):
    """Hard-max C5 with the TASK's LINEAR (non-circular) correlation.

    c(k) = sum_x h(x) * (1 - h(x+k)) * dx for k in [0, 2], h zero outside [0,2].
    Computed via zero-padded FFT exactly like the proven overlap_spectral_diagnostic,
    so its ranking matches the true objective (not the circular wrap-around value).
    """
    n = int(n_intervals); dx = 2.0 / n
    h = [0.0] * n
    for b in blocks:                        # paint the 0/1 blocks onto the grid
        s = max(0.0, min(2.0, float(b[0]))); e = max(0.0, min(2.0, float(b[1])))
        if e <= s: continue
        i0 = int(math.floor(s / dx)); i1 = min(max(int(math.ceil(e / dx)), i0 + 1), n)
        for i in range(i0, i1): h[i] = 1.0
    integral = sum(h) * dx
    a = np.zeros(n); a[:] = h
    om = 1.0 - a
    af = np.fft.rfft(a, n=2 * n)            # zero padding -> linear correlation
    of = np.fft.rfft(om, n=2 * n)
    corr = np.fft.irfft(af * np.conj(of), n=2 * n)
    c = corr[: n + 1] * dx
    c5 = float(c.max()); worst = int(np.argmax(c))
    csorted = np.sort(c)[::-1]
    gap = float(csorted[0] - csorted[1])  # how isolated the binding shift is
    return c5, round(worst * dx, 6), round(gap, 6), ..., integral

# run(): enumerate {periodic, clustered, pair} x n_bumps x widths, rescale each
# candidate to integral 1, rank by c5_hard_max (lower is better) and return
# {"best": ..., "ranked": [...], "note": "... Use overlap_spectral_diagnostic
#  on your final candidate to confirm before committing to evaluate_solution."}
```

Listing 7: Erdős: generated skills. The funding rule of `c5-eval-allocation` (round 3) and the workflow of `fast-structural-c5-search` (round 8).

```
## Funding sequence                                        -- c5-eval-allocation
1. Eval 1 -- confirm baseline.  Fund the incumbent as-is; record its hard-max c5.
2. Eval 2 -- at most one micro-polish (soft-max temperature anneal, short
   near-hard-max refinement, exact integral projection, or higher resolution),
   only with a one-sentence reason why it lowers the HARD max_k overlap.
3. Evals 3-5 -- distinct candidate families.  Each remaining evaluation funds a
   construction that differs in value scheme (sigmoid vs binary/ternary h),
   support structure (block count/widths/positions), or optimizer.
## Switch rule
After any valid evaluation, if the improvement over your best c5 is <= 1e-4
(or the run regressed or errored), do NOT micro-tune that family again.
## Reporting discipline
The training soft-max/logsumexp is a surrogate; the score is the HARD max
recomputed cleanly on the final h after projecting int h = 1 to ~1e-10.
Report the dominant shift k* and the top 2-3 c(k) values before deciding.

## Procedure                                          -- fast-structural-c5-search
1. Rank structures cheaply: block_family_sweeper at n_intervals 300-800 over
   >= 3 settings (n_bumps, spacing in {periodic, clustered, pair}, 2-4 widths).
2. Diagnose the top family: overlap_spectral_diagnostic -> worst shifts k, gap
   to the second-worst, dominant correlation mode.
3. One structural edit: change the shape of h (bump count, width, spacing),
   not the training budget, unless the diagnostic shows the grid binds.
4. Confirm once with evaluate_solution; on regression revert and take the
   next-ranked family.
```

H AUDITED REWARD-ROUTING CASE STUDY

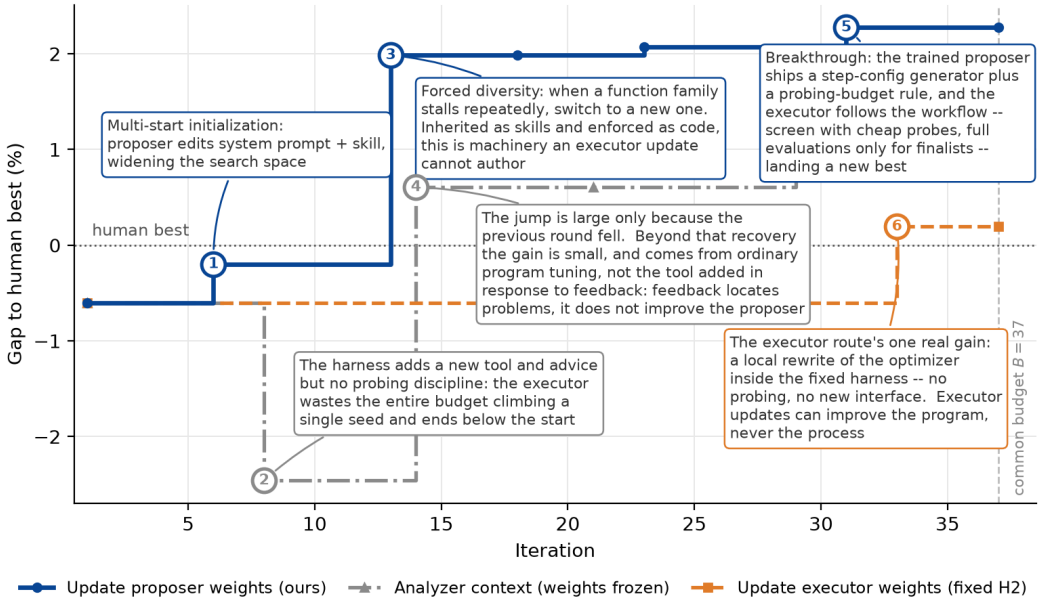


Figure 7: Autocorrelation II under three reward routes. Numbered markers identify six internally audited events. The internal record associates each event with harness files, executor trajectories, a program diff, and content hashes; these records are not part of the current public release.

According to the unreleased internal audit, event 1 adds multi-start initialization through edits to both the system prompt and skill. Event 3 inherits a forced-diversity rule from earlier skills and also instantiates it as middleware. Event 5 combines a step-configuration generator with a probing-budget rule and a screen-then- verify workflow. The corresponding unreleased program diffs associate all three events with changes in the executable search process rather than only with post-hoc descriptions.

The context route provides a useful counterexample. At event 2, a new tool and analyst advice are present, but the executor spends its budget on one seed and finishes below the starting score. Event 4 largely recovers that loss; the residual gain coincides with re-initialization and parameter-tuning changes rather than documented use of the suggested tool. The executor-update route's event 6 gain

coincides with a local optimizer rewrite inside the fixed harness. These observations are consistent with the quantitative ordering in Figure 3, but a six-event audit cannot establish that the identified mechanisms are necessary or sufficient for the score changes.