BEYOND SCRIPTED SEARCH: SAMPLE-EFFICIENT REWARD DISCOVERY VIA AGENTIC BLACK-BOX OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Designing dense reward functions for low-level reinforcement learning (RL) control remains difficult. Recent work uses large language models (LLMs) to iteratively generate and refine reward functions using policy-training feedback within scripted search algorithms. However, evaluating each candidate requires a full RL training run, making sample efficiency a central challenge for reward search on complex control tasks. To address this limitation, we propose an Agentic Reward Black-box Optimization (**ARBO**) framework, in which an LLM agent builds the search strategy at run time from an evaluation history maintained as its persistent workspace. The evaluation history comprises two components: *observations* maintained by the evaluation oracle, including candidate scores, per-term training curves, and error tracebacks; and an agent-maintained *belief* that records diagnoses and intended next steps. The agent queries both with tools and generates the next batch of reward candidates, rather than generating them in a single pass from a fixed prompt. Across four control domains, ARBO achieves gains of 29.9% in manipulation success rate and 192.8% in power-grid score over baseline means under a shared evaluation budget. Ablations examine each component's contribution and sensitivity to backbone choice.

1 INTRODUCTION

Dense reward functions are critical to reinforcement learning (RL) in low-level control. The reward is the sole optimization signal for a policy, and a task's own metric, such as success rate, is too sparse to drive learning (Makoviychuk et al., 2021; Chen et al., 2023). Dense RL rewards that measure progress toward the goal enable learning on these tasks. This process requires substantial expert effort to design appropriate reward functions (Ng et al., 1999). Large language models (LLMs) can now automate reward design. Given the task description and its environment code, an LLM writes a dense, human-readable reward function directly as executable code (Yu et al., 2023; Ma et al., 2024c; Hazra et al., 2025). Since reward quality can be assessed only through the resulting trained policy, these methods refine their proposals iteratively using the evaluation score. Rewards designed by LLMs have matched or surpassed hand-engineered rewards on these robotic tasks.

However, such iterative search is time-consuming, as each candidate reward can be evaluated only by training a policy with it in simulation. A single evaluation takes hours in control tasks for robotics and cyber-physical systems (CPS) (Makoviychuk et al., 2021; Donnot, 2020), and a full search therefore takes hundreds of hours. We measure the cost of a search by the number of reward candidates it evaluates, as this policy training dominates every other cost in the LLM search loop. Reward discovery is black-box optimization, yet existing acceleration algorithms for black-box search do not apply to it. Zeroth-order estimators (Nesterov & Spokoiny, 2017) perturb a parameter vector to recover a descent direction, but such perturbations affect only the numerical constants of a reward program, not its structure. Evolution strategies (Hansen & Ostermeier, 2001) adapt a sampling distribution over a continuous search space, but the space of reward programs is discrete. Surrogate-based optimization (Shahriari et al., 2015) maintains a belief over the objective inferred from the observations, but fitting a statistical model here would require many full RL training runs in advance. Accelerating reward discovery therefore requires improving the sample efficiency of the search strategy.

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.

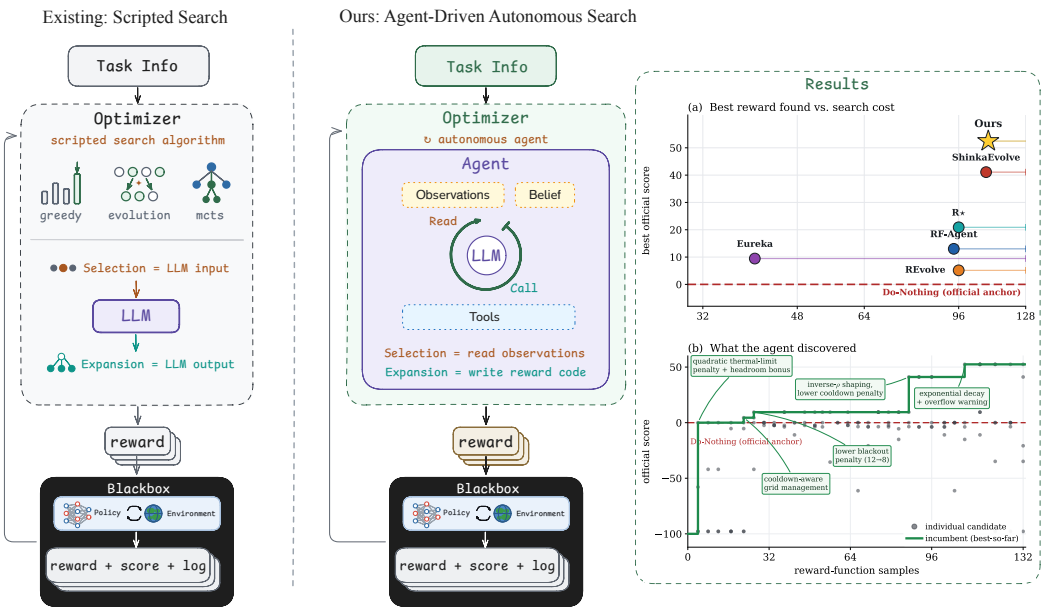


Figure 1: **Agentic Reward Black-box Optimization.** Where prior methods wrap the LLM in a scripted outer search rule, ARBO lets the agent analyze the history and propose the next candidates.

Existing automated reward design methods can be viewed as black-box optimizers, each defined by how it represents past evaluations and how it uses them to generate subsequent candidates. Reward discovery methods treat the reward program itself as the search variable, mutating expressions directly (Niekum et al., 2010) or having an LLM write it as code (Yu et al., 2023), and these LLM-based methods differ in how they retain history and generate subsequent candidates. Eureka (Ma et al., 2024c; Zeng et al., 2024) retains only the best candidate from each batch and refines it greedily. REvolve (Hazra et al., 2025; Zhao et al., 2026) maintains a population with associated scores and selects parents by score alone. RF-Agent (Gao et al., 2026) maintains a search tree with value estimates and expands the node with the highest upper confidence bound. R⋆ (Li et al., 2025) tracks the current stage and alternates structural mutation with numerical search under a predefined schedule. However, all these methods select historical candidates and use manually specified proposal rules, resulting in limited sample efficiency, as Section 3 demonstrates. Meta-black-box optimization (Ma et al., 2023) replaces such scripted rules with a meta-level policy that typically configures predefined operators, but it learns this policy by meta-training over many problem instances, which is unaffordable when every evaluation is a full RL training run.

To address this limitation, we propose ARBO, an agentic black-box optimization framework for automated reward discovery, in which an LLM agent builds the search strategy at run time from an evaluation history maintained as its persistent workspace. Prior methods specify in advance both the history they retain and the proposal rule they use, whereas ARBO retains the complete history and allows the agent to determine at runtime which records to consult and which candidates to generate next, as Figure 1 illustrates. The history comprises two components, the *observations* appended only by the evaluation oracle, which retain each candidate with its per-term curves and traceback, and the *belief* written only by the agent, which records its diagnoses and intent. This separation is inspired by Bayesian optimization: the agent maintains a revisable interpretation of past observations to guide candidate proposals, without explicitly modeling a probabilistic posterior or optimizing an acquisition function. As the proposal operator, the agent queries both components with shell and file operation tools (Yao et al., 2022; Schick et al., 2023) and generates subsequent candidates itself, using evaluation evidence rather than a predefined proposal rule. The evaluation oracle, the observations, and a fixed exploration schedule remain external to the agent, preserving evaluation as a black box beyond its control. Our agentic black-box optimization framework adapts the search to each task directly from the evaluations it has already collected.

We evaluate ARBO on robotic and CPS tasks across four control domains, including dexterous manipulation (Chen et al., 2023; Makoviychuk et al., 2021), power-grid operation (Donnot, 2020; Marot et al., 2020), legged locomotion (Mittal et al., 2025), and traffic-signal control (Ault & Sharon, 2021;

Lopez et al., 2018), against five automated LLM search methods. Under the same budget, ARBO discovers reward functions that improve the mean success rate across the ten Bi-DexHands tasks by 29.9% over the baseline mean. Power-grid evaluation further confirms that ARBO outperforms the baselines by 192.8% on average. Ablations show that the seven ablated variants lose 29.6% of the gain on average. This loss ranges from 6.0% on traffic-signal control through 19.3% on locomotion and 35.3% on manipulation to 57.7% on the power grid. These results show that ARBO outperforms scripted LLM search in efficient reward discovery on robotic and CPS tasks.

2 Related Work

Automated reward discovery. Automated methods replace the expert effort of designing dense rewards (Ng et al., 1999; Wiewiora et al., 2003; Devlin & Kudenko, 2012; Icarte et al., 2022; Ng & Russell, 2000; Christiano et al., 2017; Park et al., 2024). One-shot discovery has an LLM write the reward as code from the task description (Yu et al., 2023), without iterative refinement using policy training feedback. Iterative discovery closes this gap by revising candidates against training feedback, from genetic programming (Niekum et al., 2010) to LLM search that refines the best candidate greedily (Ma et al., 2024c; Zeng et al., 2024), evolves a population (Hazra et al., 2025; Li et al., 2025; Zhao et al., 2026), or expands a search tree or graph (Gao et al., 2026; Yao et al., 2025). Other systems extend this loop with domain randomization (Ma et al., 2024a), a relabeled replay buffer shared across candidates (Li et al., 2026b), or meta RL for adaptive CPS operations (Li et al., 2026a). RDA (Lee et al., 2026) aligns rewards with task instructions through vision-language-model judgments of rollouts, RHO (Elmaaroufi et al., 2026) searches policy code evaluated by execution, and Pan et al. (2026) refine an agent's own harness from past trajectories. These predefined search procedures may limit sample efficiency when each evaluation requires policy training, as examined in Section 3. Efficient reward search therefore requires an optimizer that analyzes the accumulated history itself and proposes high-quality candidates.

LLM agents for optimization. Reward discovery belongs to a broader class of problems in which LLMs drive gradient-free search and every candidate is scored by a black-box evaluation. Early studies regard the LLM as a variation operator inside a classical evolutionary loop and leave selection and population management to the surrounding algorithm. ELM (Lehman et al., 2023) mutates programs with diff models under a MAP-Elites archive, and LMX (Meyerson et al., 2024) performs crossover by prompting with concatenated parents and parsing the completion as offspring. FunSearch (Romera-Paredes et al., 2024) scales this operator role with an island-based program database that feeds sampled high-scoring programs into each prompt, and AlphaEvolve (Novikov et al., 2025), together with open reimplementations (Sharma, 2025; Assumpção et al., 2025), extends the recipe to entire programs and general algorithm discovery. Later systems refine the loop without changing its shape. ReEvo (Ye et al., 2024) pairs the generator LLM with a reflector LLM whose reflections serve as verbal gradients, ShinkaEvolve (Lange et al., 2026) improves sample efficiency through adaptive parent sampling and novelty-based rejection, and GEPA (Agrawal et al., 2026) mutates prompts by reflecting on execution traces under a Pareto-frontier archive. A parallel, minimalist branch dispenses with explicit evolutionary operators, as OPRO (Yang et al., 2024) prompts the LLM with the sorted trajectory of past solution-score pairs and asks for a better one directly. To further improve sample efficiency, Ma et al. (2024b) introduce differentiable simulations into discrete scientific hypotheses for continuous parameter optimization. These approaches embed LLM-based generation within externally specified search procedures that govern candidate selection and exploration (Wu et al., 2024). Meta-black-box optimization (Ma et al., 2023) learns such a procedure by training a meta-level policy over a problem distribution to configure a low-level optimizer, whereas ARBO builds the search strategy at run time on each task, with an LLM agent writing the candidates directly. The search history lives in the agent's file system, and the agent queries it with tools before writing each batch.

3 Problem Statement

In this section, we first define the reward design problem. Second, we formulate reward discovery as black-box optimization and decompose existing algorithms into a history representation and a proposal operator. We then illustrate limitations of fixed search rules on a robotics control task.

The reward design problem. A control task is a Markov decision process (MDP) $\langle S, A, T, R, \gamma, \rho_0 \rangle$ with state space S, action space A, transition dynamics T, discount γ, and initial-state distribution ρ_0. Of these, the reward function R is the sole component left to the designer. The

reward design problem (RDP) (Sorg et al., 2010) makes this explicit. The task supplies a world model $M = (S, A, T)$ with no reward, and choosing R from the space $\mathcal{R}$ of maps $S \times A \to \mathbb{R}$ completes an MDP (M, R). A fixed learning algorithm $A_{\mathcal{M}}$, the one each benchmark ships, trains under R and returns the best policy reached during training, $\pi = A_{\mathcal{M}}(R)$, and a task metric $F(\pi)$, such as success rate, scores the result. The evaluation metric is fixed, and the search varies the reward to improve policy performance. The RDP therefore asks for the reward that maximizes the task metric:

$$R^\star = \arg\max_{R \in \mathcal{R}} F\big(A_{\mathcal{M}}(R)\big). \tag{1}$$

Each candidate is executable reward code emitted by a language model p_θ conditioned on a task specification $\mathcal{T}$, comprising the environment code and its evaluation metric, and evaluating it requires training $\pi = A_{\mathcal{M}}(R)$ for a fixed step budget and measuring F. Writing $F(R) := F\big(A_{\mathcal{M}}(R)\big)$ for the induced objective on rewards, the goal is to emit code $R \sim p_\theta(\mathcal{T})$ maximizing $F(R)$.

Reward discovery as black-box optimization. The objective $F(R)$ of Equation (1) admits no analytic form. A candidate reward is evaluated by integrating it into the environment, training a policy, and measuring task performance. Reward discovery is therefore black-box optimization, and every search strategy for it proceeds by the same iteration

$$R_{t+1} \sim g\big(\cdot \mid \mathcal{T}, \phi(\mathcal{D}_t)\big), \qquad y_{t+1} = F(R_{t+1}), \qquad \mathcal{D}_{t+1} = \mathcal{D}_t \cup \{(R_{t+1}, y_{t+1})\}, \tag{2}$$

where t counts the queries issued, $\mathcal{D}_t = \{(R_i, y_i)\}_{i \le t}$ is the evaluation history, ϕ is the history representation that condenses it into the context the proposer consumes, and g is the proposal operator that turns this context into the next query. Together, (ϕ, g) form the search strategy, and we write $(\phi_t, g_t) = \Pi(\mathcal{D}_t)$ for how the strategy is built from the history at query t.

Limitations of fixed search rules. Predefined selection and proposal rules can restrict how a search uses accumulated feedback. Existing LLM reward search fixes Π by hand before the search, so every query uses the same pair. Here, ϕ *selects* which past evaluations condition the next query, by best-candidate selection, rank-based sampling, or Upper Confidence bounds applied to Trees (UCT) (Kocsis & Szepesvári, 2006), and g *expands* the selection with edits drawn from a preset operator menu at fixed rates:

$$R_{t+1} \sim \underbrace{g_{\mathcal{A}}\big(\cdot \mid \mathcal{T}, \phi(\mathcal{D}_t)\big)}_{\text{one LLM generation}}, \tag{3}$$

where the selection criteria and proposal templates are predefined, as detailed in Table 8. The templates may still condition generation on candidate code and feedback. Figure 2 measures both shortfalls on a logged MCTS-style run on DoorCloseOutward with 128 samples, configured as in Appendix C. Figure 2a marks each evaluated candidate by whether any later proposal reused it. The score-based rule carries forward only a handful, and the code, the per-term reward curves, and the tracebacks of the other 79% enter no later prompt in this run. Pooled over seven Bi-DexHands tasks, 77% of the 684 candidates neither improve the incumbent nor enter a later prompt. Failed and low-scoring candidates are therefore unavailable for explicit reuse when the next candidate is written. Figure 2b shows limited improvement in candidate quality over the budget, with search performance plateauing below the maximum task score in this run. Learning Π instead (Ma et al., 2023) would require meta-training over many problem instances, where each is evaluated by full RL training runs.

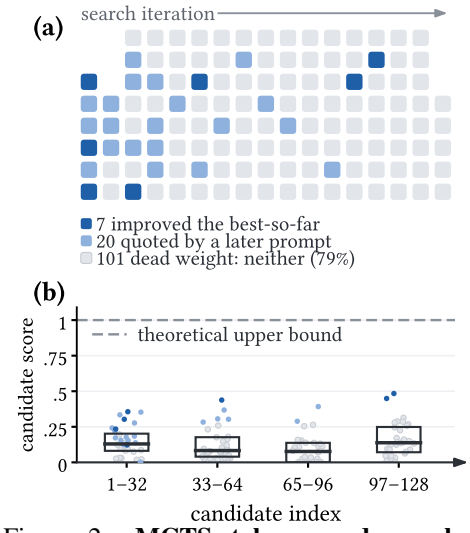


Figure 2: **MCTS-style reward search: (a)** Selection coverage **(b)** Candidate score.

4 System Design

This section presents the system design of ARBO. We first specify the history representation and the proposal operator, with which the agent reads the history and writes reward candidates. Second, we specify the oracle and the exploration schedule, which remain outside the agent. We then assemble the four components into the complete optimization loop.

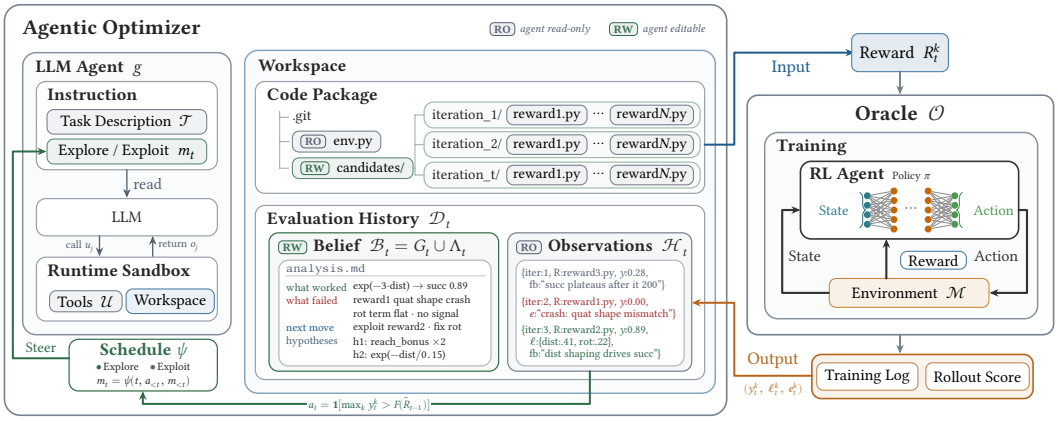


Figure 3: **ARBO framework design.** The agent queries the history $\mathcal{D}_t$ with tools and commits reward candidates, while the oracle owns the observations and the schedule issues the prompt m_t.

System overview. ARBO performs the black-box optimization of Equation (2) with four components, the history $\mathcal{D}_t$, the proposal operator g, the evaluation oracle $\mathcal{O}$, and the schedule ψ, as Figure 3 shows. In each round, ARBO performs scheduling, proposal, and evaluation in sequence:

$$m_t = \psi\big(t, a_{<t}, m_{<t}\big), \qquad \{R_t^k\}_{k=1}^{N_t} \sim g\big(\cdot \mid \mathcal{T}, \mathcal{D}_t, m_t\big), \qquad \big(y_t^k, \ell_t^k, e_t^k\big) = \mathcal{O}\big(R_t^k\big), \tag{4}$$

where t now indexes rounds, m_t is the steering prompt of round t, $a_{<t} = (a_1, \ldots, a_{t-1})$ the acceptance record, $m_{<t}$ the past steering prompts, $\{R_t^k\}_{k=1}^{N_t}$ the batch of candidates the agent commits, and (y_t^k, ℓ_t^k, e_t^k) the score, the per-term curves and the traceback of candidate R_t^k. The schedule emits the steering prompt from the acceptance record, the agent generates the batch conditioned on the task specification and the history, and the oracle trains and scores every candidate, appending the results to the history. Thus, the agent builds (ϕ_t, g_t) of Section 3 at every round, selecting what it reads and expanding it into the next candidates.

History representation. We remove compression from the stored history, and $\mathcal{D}_t$ retains the evaluation history in full, in two parts:

$$\mathcal{D}_t = \underbrace{\mathcal{H}_t}_{\text{observations}} \cup \underbrace{\mathcal{B}_t}_{\text{belief}}, \qquad \mathcal{B}_t = G_t \cup \Lambda_t, \tag{5}$$

where $\mathcal{H}_t$ are the observations maintained outside, $\mathcal{B}_t$ the belief updated by the agent itself, G_t the agent's rolling notes, and Λ_t the lineage of its edits.

The observations $\mathcal{H}_t$ are the oracle's side of the history, the evaluation history of Equation (2) kept whole and grown only by union:

$$\mathcal{H}_t = \{(R_r^k, y_r^k, \ell_r^k, e_r^k)\}_{r<t}, \qquad \mathcal{H}_{t+1} = \mathcal{H}_t \cup h_t, \qquad h_t = \big\{\big(R_t^k, y_t^k, \ell_t^k, e_t^k\big)\big\}_{k=1}^{N_t}, \tag{6}$$

where $r < t$ runs over the rounds before t and k over their candidates, each entry pairs R_r^k with the score y_r^k, the per-term curves ℓ_r^k and the traceback e_r^k, and h_t is the batch of records the round appends. The oracle alone maintains them, and no agent action can revise them.

The belief $\mathcal{B}_t$ is the agent's side, and only the agent writes it:

$$G_{t+1} \sim \mathrm{LLM}_\theta\big(\cdot \mid G_t,\ \mathcal{H}_t\big), \qquad \Lambda_{t+1} = \Lambda_t \cup \big\{(R_t^k,\ p_t)\big\}_{k=1}^{N_t}. \tag{7}$$

Each round the agent overwrites G_t with its diagnoses and its intent, kept short since the full observations stay in $\mathcal{H}_t$. The belief records hypotheses and search decisions that guide subsequent proposals, serving as an analogue of a posterior in Bayesian optimization. The candidates are committed into Λ_t with their parent commit p_t, on the best line when refining and on a fresh branch when exploring, a placement that records the agent's judgment of which line is worth extending and hence belongs to the belief.

Proposal operator. We implement g as one full run of an autonomous agent, the frozen model LLM_θ driving a sandboxed tool set $\mathcal{U}$ of bash, file operations and search (Yao et al., 2022; Schick

et al., 2023), under a fixed instruction file, as Appendix G details. The run works in the workspace of Figure 3, where the environment code of $\mathcal{T}$ sits read-only in the code package, the candidates of each round are committed into it under git, the observations are the appended evaluation log, and the belief is the analysis file. The run conditions on these files through its tool calls:

$$u_{j+1} \sim \mathrm{LLM}_\theta\big(\cdot \mid c_t,\, \tau_j\big), \qquad o_{j+1} = \mathcal{U}\big(u_{j+1},\, \mathcal{T},\, \mathcal{D}_t\big), \qquad \tau_j = (u_1, o_1, \ldots, u_j, o_j), \tag{8}$$

where c_t is the fixed context of round t, holding the instruction, the task description and the steering prompt m_t of Equation (11), u_j a tool call, and o_j its observation of the workspace files. A typical run reads the recorded scores, inspects policy training curves for the incumbent reward, checks the traceback of a failed candidate, and writes the next batch of candidates.

The run ends with a tool call that commits the batch of candidates into the code package, and chaining the steps of Equation (8) gives the proposal operator of Equation (4):

$$g\big(\{R_t^k\}_{k=1}^{N_t},\, \tau \mid \mathcal{T}, \mathcal{D}_t, m_t\big) = \prod_{j \geq 0} \mathrm{LLM}_\theta\big(u_{j+1} \mid c_t,\, \tau_j\big), \tag{9}$$

where the product runs over the tool calls of τ, and the last of them commits the batch $\{R_t^k\}_{k=1}^{N_t}$. The batch results from tool interactions with the workspace, rather than a single generation call. Between rounds, the agent retains state through the history $\mathcal{D}_t$, which informs subsequent proposals.

Oracle. In black-box optimization, each candidate is evaluated by an oracle, and a classical zeroth-order oracle returns only the scalar objective value (Nesterov & Spokoiny, 2017). Our oracle returns richer feedback:

$$\mathcal{O}(R) = \big(F\big(A_{\mathcal{M}}(R)\big),\, \ell(R),\, e(R)\big), \qquad \mathcal{O} \notin \mathcal{U}, \tag{10}$$

where $A_{\mathcal{M}}$ trains one policy per committed candidate under the domain's own published configuration, F is the task's acceptance metric, ℓ the per-term reward curves over training, e the traceback when a candidate fails to run, and the triple is appended to $\mathcal{H}_t$. The oracle runs outside the agent's environment, and no tool in $\mathcal{U}$ can invoke it.

Exploration-Exploitation Schedule. The exploration-exploitation schedule is controlled externally. An unconstrained agent may overexploit the incumbent, and its belief provides no calibrated uncertainty estimates to guide exploration. The schedule ψ therefore imposes the trade-off as the only scripted decision rule in the system, and it reads only the acceptance record and its own past prompts, carrying no knowledge of what the candidates contain:

$$a_t = \mathbf{1}\Big[\max_k y_t^k > F\big(\hat{R}_{t-1}\big)\Big], \qquad m_t = \psi\big(t, a_{<t}, m_{<t}\big), \tag{11}$$

where $a_t = 1$ indicates that the round produces a candidate with a higher score than the incumbent reward $\hat{R}_{t-1}$. The steering prompt takes one of two fixed values, m_{exploit} and m_{explore}, each injected into the agent's context, directing the agent to extend $\hat{R}_{t-1}$ or to open a fresh line from scratch, and Appendix G shows the injected text. Specifically, the schedule opens with n_0 rounds of m_{explore}, holds m_{exploit} once an incumbent exists, and returns to m_{explore} for at most ν rounds after $a_t = 0$ holds for μ consecutive rounds.

Optimization loop. The search starts from the bootstrap code base with an empty history and repeats the round of Equation (4) under the schedule, until the sample budget is spent. The search returns the incumbent reward, the highest-scoring candidate in the history. Algorithm 1 in Appendix E shows the full reward discovery procedure of ARBO.

5 Experiments

This section describes the four control domains and the five automated baselines, and reports the comparison results. It then presents ablation studies on each component and backbone models.

5.1 Experiment Environments

We test ARBO in four control domains, each with its own acceptance metric. Appendix A gives the score definitions and the sources of the task descriptions.

Dexterous manipulation. The ten dual-arm Bi-DexHands tasks (Chen et al., 2023) in Isaac Gym (Makoviychuk et al., 2021) cover handover and catching, articulated-object manipulation, and

pick-and-place, stacking and pouring. Success is binary, and the score is the success rate. Appendix A lists all ten tasks.

Power-grid operation. The Case14 sandbox of Grid2Op (Donnot, 2020) is scored by the L2RPN scorer over 16 held-out scenarios. Leading entries wrap RL in expert heuristics and simulator lookahead (Marot et al., 2020; 2021), whereas we keep the task pure RL and place the whole operating objective on the reward.

Locomotion. The four Isaac Lab (Mittal et al., 2025) tasks ask the ANYmal-D quadruped and the Cassie biped to track a commanded planar velocity and yaw rate on flat and on rough terrain, and the score charges tracking error and falls, where 0 is perfect tracking.

Traffic-signal control. The RESCO benchmark (Ault & Sharon, 2021) runs SUMO (Lopez et al., 2018) on real road networks with measured demand, one independent learner per intersection, over four networks. The score is the negated official average delay.

5.2 Baselines

We compare against five automated LLM search methods, each an instance of Equation (3): a *selection* rule fixed before the search begins decides which past candidates condition the next query, and a fixed *expansion* rule $g_{\mathcal{A}}$ turns them into new candidates. The baseline methods include:

Eureka (Ma et al., 2024c) selects the single best candidate of the previous batch and expands it with one improvement template.

R⋆ (Li et al., 2025) reflects on the global best under a hint from {improve, remove, add} or crosses two rank-sampled parents, alternating with a parameter-alignment stage.

REvolve (Hazra et al., 2025) maintains an evolving population of reward functions, selects parents from it by fitness, and expands them with mutation and crossover operations. We use its fully automated variant, where feedback comes from the environment rather than a human.

ShinkaEvolve (Lange et al., 2026) selects a score-weighted parent plus one archive and one top-k inspiration and expands with a patch type drawn at fixed rates, testing whether a strong general program-evolution searcher closes the gap.

RF-Agent (Gao et al., 2026) selects a node of its search tree by UCT, of which at most four enter any prompt, and expands it with a fixed menu of five actions.

Settings. We compare all methods at 72 reward samples on locomotion and 128 on the other three domains. Within each domain, methods share the evaluation budget, the `MiniMax-M2.7` backbone, and the policy-training configuration. We report *best-so-far* score against *reward samples*, and ARBO is labeled *Ours* in every table and figure. Appendix B gives the full training settings.

5.3 Experiment Results

Advantages of the agent under a shared budget. Table 1 reports each method's best-so-far success rate under the shared budget, where *Sparse* uses the success indicator as the reward under the same policy training configuration as searched rewards. ARBO attains the best score on most tasks. It also never ranks below second among the automated methods on any of the ten tasks. Baseline performance varies across tasks. Individual baselines excel on some tasks but perform poorly on others, and none maintains an advantage across the suite. ARBO maintains high success rates across tasks. Every method shares the same backbone and the same evaluation budget, and Appendix B reports their LLM inference costs.

Table 1: Bi-DexHands success rate within the 128-sample budget, where **bold** and underline mark the best and second-best automated search methods and runs short of the budget are reported at the samples completed.

Method	*Easy*				*Medium*			*Hard*			
	Swing Cup	Lift Underarm	Door Close	Over	Grasp Place	Block Stack	Bottle Cap	Kettle	Catch Underarm	Catch Abreast	**Avg**
Sparse	0.01	0.00	0.06	0.00	0.00	0.02	0.03	0.02	0.00	0.00	0.014
REvolve	0.920	0.906	0.606	0.947	0.510	0.323	0.695	0.970	0.298	0.372	0.655
Eureka	0.994	0.889	0.972	0.939	0.527	0.529	0.873	0.965	0.676	0.558	0.792
RF-Agent	0.984	0.930	0.484	0.925	0.366	0.170	0.837	0.994	0.819	0.560	0.707
R⋆	**0.999**	0.918	0.580	0.970	0.339	0.352	0.434	0.958	**0.850**	0.000	0.640
ShinkaEvolve	0.879	0.738	0.775	0.878	0.330	**0.917**	0.609	0.986	0.753	0.541	0.741
Ours	0.994	**0.979**	**1.000**	**0.971**	**0.984**	0.812	**0.988**	**0.999**	0.849	**0.603**	**0.918**

Better rewards with fewer evaluations. Figure 4 groups the ten tasks by difficulty and plots group-mean best-so-far success rates against reward samples. ARBO achieves the highest final score in every group, and its score at 64 samples already exceeds every baseline's final score. The gap is largest on the medium group, where it grows throughout the budget. On the hard group, ARBO reaches its plateau within about 30 samples, while the baselines close most of the gap by the end of the budget. On the easy group, every method ends above 0.8 and the final gap is small.

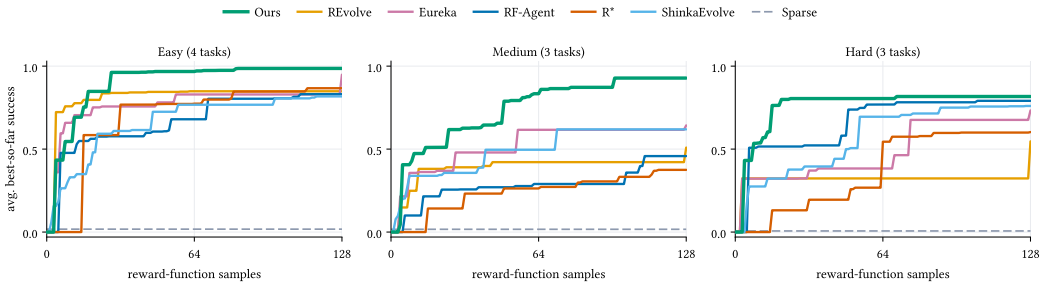


Figure 4: Bi-DexHands sample efficiency by task difficulty, as group-mean best-so-far success against reward samples, with the group-mean *Sparse* reference dashed.

Autonomy generates better candidates. Figure 5 summarizes the quality of all logged candidates. In Figure 5a, the success rate distribution of ARBO's candidates dominates those of the fixed-rule baselines. In Figure 5b, a larger fraction of its candidates exceeds each quality threshold than for any baseline. In Figure 5c, its mean candidate quality increases most between the first and last budget quartiles, while even the strongest fixed-rule baseline improves less and the others still less. Thus, ARBO proposes higher-quality candidates on average, allocating fewer costly evaluations to low-quality candidates. This improvement during search stems from the agent's autonomy. The agent generates candidates after analyzing accumulated records, allowing subsequent batches to incorporate feedback from earlier failures. The predefined proposal operator $g_{\mathcal{A}}$ cannot match this improvement in candidate quality.

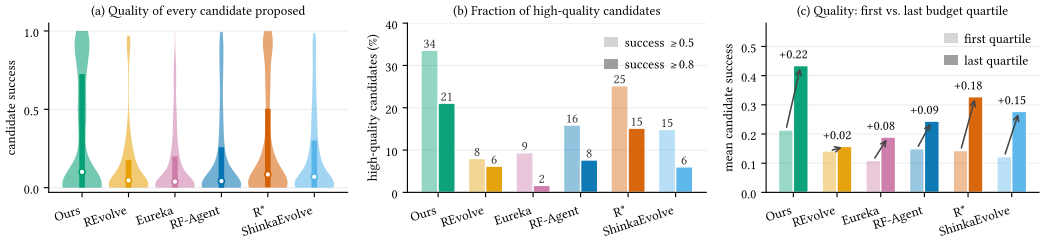


Figure 5: Bi-DexHands candidate statistics per method: (a) success of all proposed candidates, (b) fractions clearing 0.5 and 0.8, (c) mean success in the first versus last budget quartile.

The advantage widens on sparser-reward tasks. Figure 6 tracks the search on the grid, locomotion, and traffic tasks. On traffic, with delay feedback at every step, all methods end at nearly the same level. On locomotion, ARBO pulls ahead early and keeps the lead, though the baselines recover most of the gain. On the power grid, the sparsest setting, our reward scores highest and completes the

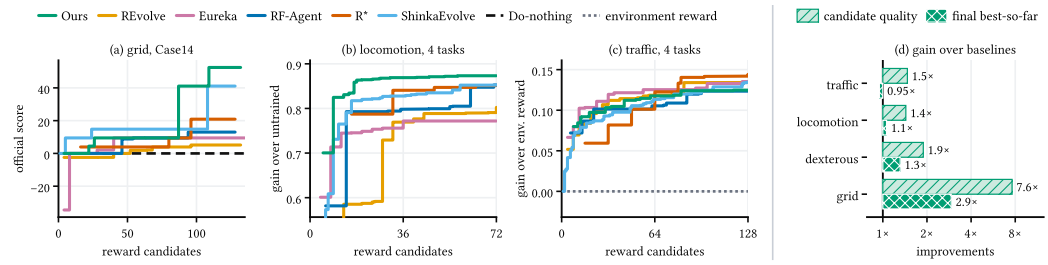


Figure 6: Search progress on (a) Case14 by official score, and by mean relative gain over (b) the untrained policy on locomotion and (c) the environment reward on traffic. (d) ARBO's gain over the baseline mean in candidate quality and final best-so-far, domains from dense to sparse feedback.

most scenarios, and most baselines complete none. Figure 6d adds manipulation and orders all domains from dense to sparse feedback. Candidate quality leads everywhere, and the final best-so-far dips to $0.95\times$ on traffic and reaches $2.9\times$ on the grid. The advantage is noticeable where feedback is sparse and unsuccessful candidates require diagnosis beyond their evaluation scores. Appendix D.2 analyzes the grid states, and Appendices D.3 and D.4 report the locomotion and traffic runs.

5.4 Ablations

History Representation. To evaluate access to historical information, we ablate each history component and the access policy. *w/o belief* removes the self-maintained belief (Shinn et al., 2023), and *w/o observations* keeps only the bounded context of memory-managed agents (Packer et al., 2023). *retrieval (embed)* and *retrieval (GA)* select observations for the agent using embedding similarity (Lewis et al., 2020) or the retrieval rule of Generative Agents (GA) (Park et al., 2023). As shown in Figure 7a, nearly every restriction on history access reduces search performance. Removing observations reduces performance more than removing the belief on every task, indicating their central role. Retrieval usually costs less than removing the observations but still limits performance gains. The search thus draws on evidence that score-based or similarity-based selection would omit.

Exploration vs. Exploitation. To evaluate the schedule ψ, we separately disable exploration and exploitation (Sutton & Barto, 1998) in the scheduling rule. *w/o explore* always injects m_{exploit} and refines reward candidates from the best attempts, while *w/o exploit* always injects m_{explore} and restarts every round. *codex* removes the framework entirely and gives one stock coding agent the same task, backbone, and sample budget. As shown in Figure 7b, the performance loss from using either strategy alone. On the easier traffic task every variant matches the full system, since dense feedback lets refinement and restarts converge on the same designs. On the harder power grid, *w/o explore* and *codex* give up most of the gain. This indicates that the schedule is dispensable where feedback is dense and essential where it is sparse.

LLM Backbone. To evaluate backbone effects, we run ARBO with four alternative models. Figure 7c reports the results. Scores are normalized using reference-reward performance and the highest backbone score for each task as shown in Appendix D.5. The non-reasoning backbone yields almost no performance gain. On most tasks, it exhausts the budget without a reward that supports successful policy training or outperforms the reference reward. Therefore, the proposed framework is not suited to weak backbones.

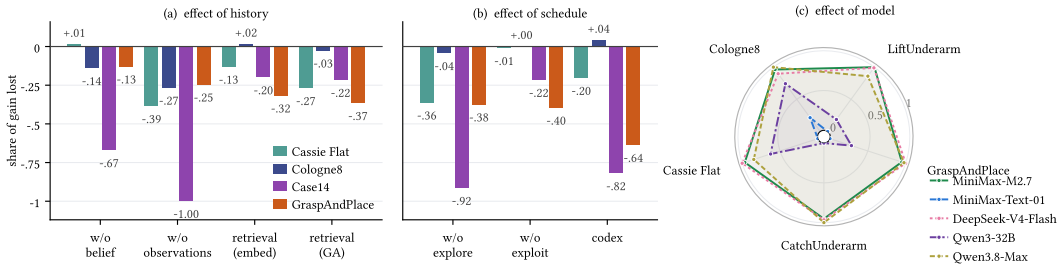


Figure 7: Performance loss from ablating (a) the history, by removing the belief, the observations, or interposing a retriever, (b) the schedule, by using only exploration or exploitation or removing the framework, and (c) normalized score by backbone.

6 Limitation & Conclusion

Limitation. ARBO reasons over a purely textual history of scores, per-term curves, and tracebacks. Some failures are apparent only from non-textual observations, such as a policy achieving a high score through unnatural movements (Wang et al., 2024) or a data center judged by its thermal fields (Cao et al., 2026). Extending the framework with multimodal reasoning is left to future work.

Conclusion. This paper proposes ARBO, an agentic black-box optimization framework for automated reward discovery, in which an LLM agent builds the search strategy at run time from the evaluation history. The history is kept in two parts, the observations the oracle appends with the evaluation results and the belief the agent maintains with its diagnoses and intent. Experiments across diverse control tasks show performance gains over the evaluated baselines, and ablations support the contributions of the proposed components.

AI USE STATEMENT

We used generative AI tools to aid or polish writing, and for retrieval and discovery, which covers sourcing information and identifying relevant literature when surveying related work. We have reviewed all AI-assisted work. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

Section 4 and Appendix E describe ARBO and its full search procedure, and Appendix G gives all prompts. Appendices A–C specify the tasks, training configurations, seeds, sample budgets, schedule constants, and baseline settings. Our code is available at `https://anonymous.4open.science/r/arbo-30C6/`.

ETHICS STATEMENT

This work involves no human subjects and no personal data, and all benchmarks are public simulators used under the licenses in Appendix F.

REFERENCES

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

Henrique Assumpção, Diego Ferreira, Leandro Campos, and Fabricio Murai. CodeEvolve: an open source evolutionary coding agent for algorithmic discovery and optimization. *arXiv preprint arXiv:2510.14150*, 2025.

James Ault and Guni Sharon. Reinforcement learning benchmarks for traffic signal control. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

Zhiwei Cao, Minghao Li, Feng Ling, Jimin Jia, Yonggang Wen, Jianxiong Yin, and Simon See. Transforming future data center operations and management via physical AI. *IEEE Intelligent Systems*, 2026.

Yuanpei Chen, Yiran Geng, Fangwei Zhong, Jiaming Ji, Jiechuang Jiang, Zongqing Lu, Hao Dong, and Yaodong Yang. Bi-DexHands: Towards human-level bimanual dexterous manipulation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 46(5):2804–2818, 2023.

Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. *Advances in Neural Information Processing Systems*, 30, 2017.

Sam Michael Devlin and Daniel Kudenko. Dynamic potential-based reward shaping. In *11th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2012)*, pp. 433–440. IFAAMAS, 2012.

Benjamin Donnot. Grid2op-a testbed platform to model sequential decision making in power systems. *GitHub repository*, 2020.

Karim Elmaaroufi, Justin Svegliato, Sarunas Kalade, Graham Schelle, Sanjit A Seshia, and Matei Zaharia. RHO: Your coding agent is secretly a roboticist. *arXiv preprint arXiv:2606.16458*, 2026.

Ning Gao, Xiuhui Zhang, Xingyu Jiang, Mukang You, Mohan Zhang, and Yue Deng. RF-Agent: automated reward function design via language agent tree search. *Advances in Neural Information Processing Systems*, 38:172532–172577, 2026.

Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9(2):159–195, 2001.

Rishi Hazra, Alkis Sygkounas, Andreas Persson, Amy Loutfi, and Pedro Zuidberg Dos Martires. REvolve: Reward evolution with large language models using human feedback. In *International Conference on Learning Representations*, volume 2025, pp. 101949–101990, 2025.

Rodrigo Toro Icarte, Toryn Q Klassen, Richard Valenzano, and Sheila A McIlraith. Reward machines: Exploiting reward function structure in reinforcement learning. *Journal of Artificial Intelligence Research*, 73:173–208, 2022.

Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *European Conference on Machine Learning*, pp. 282–293. Springer, 2006.

Robert Lange, Yuki Imajuku, and Edoardo Cetin. ShinkaEvolve: Towards open-ended and sample-efficient program evolution. In *International Conference on Learning Representations*, volume 2026, pp. 74026–74078, 2026.

Hojoon Lee, Ajay Subramanian, Ben Abbatematteo, Vijay Veerabadran, Pedro Matias, Karl Ridgeway, and Nitin Kamra. Reward design agent for reinforcement learning. *Reinforcement Learning Journal*, 7, 2026.

Joel Lehman, Jonathan Gordon, Shawn Jain, Kamal Ndousse, Cathy Yeh, and Kenneth O Stanley. Evolution through large models. In *Handbook of Evolutionary Machine Learning*, pp. 331–366. Springer, 2023.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive NLP tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

Minghao Li, Ruihang Wang, Rui Tan, and Yonggang Wen. DCoPilot: Generative AI-empowered policy adaptation for dynamic data center operations. In *2026 ACM/IEEE 17th International Conference on Cyber-Physical Systems (ICCPS)*, pp. 189–200. IEEE, 2026a.

Pengyi Li, Jianye Hao, Hongyao Tang, Yifu Yuan, Jinbin Qiao, Zibin Dong, and Yan Zheng. R*: Efficient reward design via reward structure evolution and parameter alignment optimization with large language models. In *ICML*, 2025.

Pengyi Li, Hongyao Tang, Jinbin Qiao, Yan Zheng, and Jianye Hao. LaRes: evolutionary reinforcement learning with LLM-based adaptive reward search. *Advances in Neural Information Processing Systems*, 38:23270–23298, 2026b.

Pablo Alvarez Lopez, Michael Behrisch, Laura Bieker-Walz, Jakob Erdmann, Yun-Pang Flötteröd, Robert Hilbrich, Leonhard Lücken, Johannes Rummel, Peter Wagner, and Evamarie Wießner. Microscopic traffic simulation using SUMO. In *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, pp. 2575–2582. IEEE, 2018.

Jason Ma, William Liang, Hung-Ju Wang, Yuke Zhu, Linxi Fan, Osbert Bastani, and Dinesh Jayaraman. DrEureka: Language model guided sim-to-real transfer. In *Robotics: Science and Systems*, 2024a.

Pingchuan Ma, Tsun-Hsuan Wang, Minghao Guo, Zhiqing Sun, Joshua B Tenenbaum, Daniela Rus, Chuang Gan, and Wojciech Matusik. LLM and simulation as bilevel optimizers: A new paradigm to advance physical scientific discovery. *arXiv preprint arXiv:2405.09783*, 2024b.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Jim Fan, et al. Eureka: Human-level reward design via coding large language models. In *International Conference on Learning Representations*, volume 2024, pp. 26516–26560, 2024c.

Zeyuan Ma, Hongshu Guo, Jiacheng Chen, Zhenrui Li, Guojun Peng, Yue-Jiao Gong, Yining Ma, and Zhiguang Cao. MetaBox: A benchmark platform for meta-black-box optimization with reinforcement learning. *Advances in Neural Information Processing Systems*, 36:10775–10795, 2023.

Denys Makoviichuk and Viktor Makoviychuk. rl-games: A high-performance framework for reinforcement learning. *GitHub repository*, 2021.

Viktor Makoviychuk, Lukasz Wawrzyniak, Yunrong Guo, Michelle Lu, Kier Storey, Miles Macklin, David Hoeller, Nikita Rudin, Arthur Allshire, Ankur Handa, et al. Isaac Gym: High performance GPU-based physics simulation for robot learning. *arXiv preprint arXiv:2108.10470*, 2021.

Antoine Marot, Benjamin Donnot, Camilo Romero, Balthazar Donon, Marvin Lerousseau, Luca Veyrin-Forrer, and Isabelle Guyon. Learning to run a power network challenge for training topology controllers. *Electric Power Systems Research*, 189:106635, 2020.

Antoine Marot, Benjamin Donnot, Gabriel Dulac-Arnold, Adrian Kelly, Aidan O'Sullivan, Jan Viebahn, Mariette Awad, Isabelle Guyon, Patrick Panciatici, and Camilo Romero. Learning to run a power network challenge: a retrospective analysis. In *NeurIPS 2020 Competition and Demonstration Track*, pp. 112–132. PMLR, 2021.

Elliot Meyerson, Mark J Nelson, Herbie Bradley, Adam Gaier, Arash Moradi, Amy K Hoover, and Joel Lehman. Language model crossover: Variation through few-shot prompting. *ACM Transactions on Evolutionary Learning and Optimization*, 4(4):1–40, 2024.

Mayank Mittal, Pascal Roth, James Tigue, Antoine Richard, Octi Zhang, Peter Du, Antonio Serrano-Munoz, Xinjie Yao, René Zurbrügg, Nikita Rudin, et al. Isaac Lab: A GPU-accelerated simulation framework for multi-modal robot learning. *arXiv preprint arXiv:2511.04831*, 2025.

Yurii Nesterov and Vladimir Spokoiny. Random gradient-free minimization of convex functions. *Foundations of Computational Mathematics*, 17(2):527–566, 2017.

Andrew Y Ng and Stuart Russell. Algorithms for inverse reinforcement learning. In *ICML*, pp. 663–670, 2000.

Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, pp. 278–287, 1999.

Scott Niekum, Andrew G Barto, and Lee Spector. Genetic programming for reward function search. *IEEE Transactions on Autonomous Mental Development*, 2(2):83–90, 2010.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G Patil, Ion Stoica, and Joseph E Gonzalez. MemGPT: Towards LLMs as operating systems. *arXiv preprint arXiv:2310.08560*, 2023.

Wenbo Pan, Shujie Liu, Chin-Yew Lin, Jingying Zeng, Xianfeng Tang, Xiangyang Zhou, Yan Lu, and Xiaohua Jia. Evolving agents in the dark: Retrospective harness optimization via self-preference. *arXiv preprint arXiv:2606.05922*, 2026.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, pp. 1–22, 2023.

Younghyo Park, Gabriel B Margolis, and Pulkit Agrawal. Position: Automatic environment shaping is the next frontier in RL. In *Forty-first International Conference on Machine Learning*, 2024.

Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-Baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

Nikita Rudin, David Hoeller, Philipp Reist, and Marco Hutter. Learning to walk in minutes using massively parallel deep reinforcement learning. In *Conference on Robot Learning*, pp. 91–100. PMLR, 2022.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551, 2023.

Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. Taking the human out of the loop: A review of Bayesian optimization. *Proceedings of the IEEE*, 104(1): 148–175, 2015.

Asankhaya Sharma. OpenEvolve: an open-source evolutionary coding agent. *GitHub repository*, 2025.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652, 2023.

Jonathan Sorg, Richard L Lewis, and Satinder Singh. Reward design via online gradient ascent. *Advances in Neural Information Processing Systems*, 23, 2010.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT Press, 1998.

Yufei Wang, Zhanyi Sun, Jesse Zhang, Zhou Xian, Erdem Biyik, David Held, and Zackory Erickson. RL-VLM-F: Reinforcement learning from vision language foundation model feedback. *arXiv preprint arXiv:2402.03681*, 2024.

Eric Wiewiora, Garrison W Cottrell, and Charles Elkan. Principled methods for advising reinforcement learning agents. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pp. 792–799, 2003.

Xingyu Wu, Sheng-hao Wu, Jibin Wu, Liang Feng, and Kay Chen Tan. Evolutionary computation in the era of large language model: Survey and roadmap. *IEEE Transactions on Evolutionary Computation*, 29(2):534–554, 2024.

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

Changwei Yao, Xinzi Liu, Chen Li, and Marios Savvides. Reward evolution with graph-of-thoughts: A bi-level language model framework for reinforcement learning. *arXiv preprint arXiv:2509.16136*, 2025.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. ReEvo: Large language models as hyper-heuristics with reflective evolution. *Advances in Neural Information Processing Systems*, 37:43571–43608, 2024.

Wenhao Yu, Nimrod Gileadi, Chuyuan Fu, Sean Kirmani, Kuang-Huei Lee, Montse Gonzalez Arenas, Hao-Tien Lewis Chiang, Tom Erez, Leonard Hasenclever, Jan Humplik, et al. Language to rewards for robotic skill synthesis. *arXiv preprint arXiv:2306.08647*, 2023.

Yuwei Zeng, Yao Mu, and Lin Shao. Learning reward for robot skills using large language models via self-alignment. *arXiv preprint arXiv:2405.07162*, 2024.

Zhikai Zhao, Chuanbo Hua, Federico Berto, Zihan Ma, Kanghoon Lee, Jiachen Li, and Jinkyoo Park. EvoNav: Evolutionary reward function design for robot navigation with large language models. *arXiv preprint arXiv:2605.11859*, 2026.

A ENVIRONMENTS

Every method reads the same natural-language description of each task. We do not reproduce these descriptions and instead cite the source that documents each benchmark.

Bi-DexHands. Bi-DexHands (Chen et al., 2023) places two shadow hands in Isaac Gym and asks them to cooperate, passing an object between hands, operating a tool, or placing a grasped object. Success is defined by a binary predicate over distances and orientations. The score therefore provides no partial credit. The dense reward supplies the learning signal. Figure 8 illustrates the ten tasks, whose descriptions follow the original Bi-DexHands paper (Chen et al., 2023).

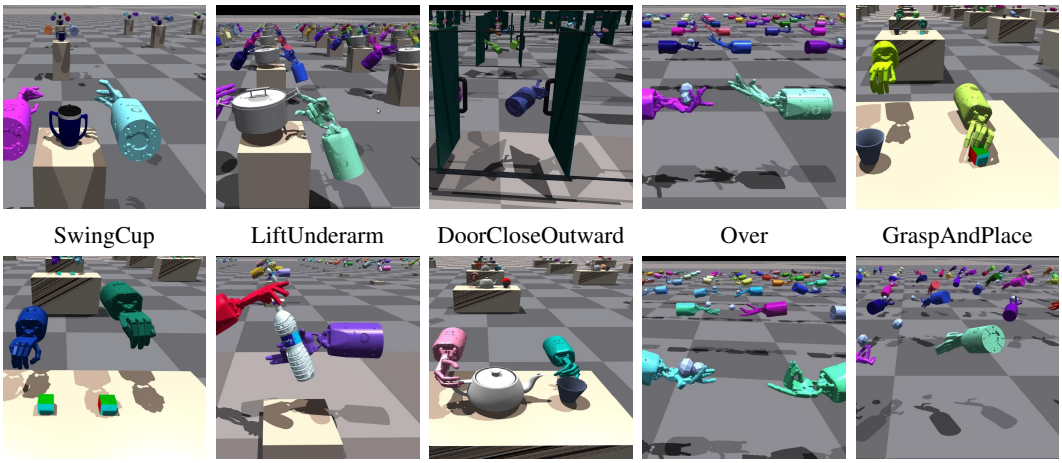


Figure 8: Bi-DexHands task environments.

Grid2Op. Grid2Op (Donnot, 2020) simulates the real-time operation of a power transmission grid, held through four weeks of recorded demand by reconfiguring busbars, switching lines and redispatching generators. The score is the competition's operating cost rather than a success flag, pricing the grid survives and every step it does not. We use the L2RPN Case14 sandbox of Figure 9, the IEEE case-14 benchmark (Marot et al., 2020), which realizes the MDP $\langle S, A, T, R, \gamma, \rho_0 \rangle$ as follows.

- **State** S**.** The official Grid2Op state $o_t \in \mathbb{R}^{538}$: generator and load injections, per-line flows, loadings and status, overflow counters, cooldown timers, maintenance schedules, calendar features, and every element's busbar assignment.
- **Actions** A**.** The $|A| = 273$ competition controls: do-nothing (1), unitary single-substation bus reassignment (178), line disconnection and reconnection (40), unitary redispatch (24) and renewable curtailment (30).
- **Transitions** T**.** Deterministic given the scenario: the simulator applies a_t, replays the next recorded injections, and re-solves the power flow. Writing $\rho_{\ell,t} = |f_{\ell,t}|/\bar{f}_\ell$ for the loading of line ℓ against its thermal limit, a line above 1 for three consecutive steps or immediately above 2 is disconnected and its flow redistributes onto neighbors, and the resulting cascade typically blacks out the grid at step $H_{\text{alive}} \leq H$.
- **Initial state** ρ_0 **and horizon.** The initial state is the start of a scenario, which specifies a fixed sequence of injections over $H = 8{,}064$ steps at five-minute resolution. Evaluation on the 16 held-out scenarios use deterministic replays of these sequences.
- **Reward** R**.** The object of search. Policies are trained with discount $\gamma = 0.99$, and candidate selection uses the external evaluation score defined below.

Acceptance uses the L2RPN competition score (Marot et al., 2020), which at every survived step charges the network losses plus the dispatch corrections actually applied, at the current marginal price,

$$c_t \;=\; p_t\,\Delta\Big(\underbrace{\textstyle\sum_g P^{\text{gen}}_{g,t} - \sum_d P^{\text{load}}_{d,t}}_{\text{network losses}} + \underbrace{\textstyle\sum_g \big|P^{\text{redisp}}_{g,t}\big|}_{\text{redispatch}} + \underbrace{\big(P^{\text{curt}}_{t} - P^{\text{curt}}_{t-1}\big)}_{\text{curtailment change}}\Big), \tag{12}$$

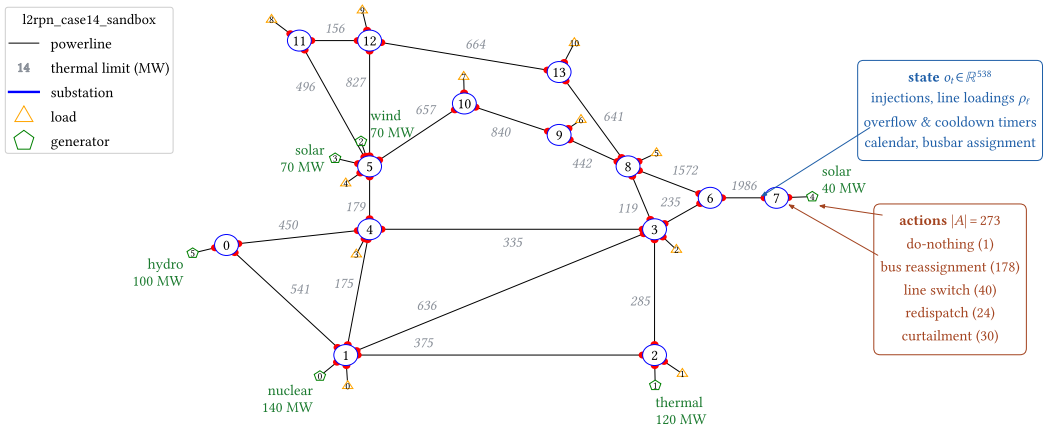


Figure 9: L2RPN Case14 environment.

Here $P^{\text{gen}}_{g,t}$ and $P^{\text{load}}_{d,t}$ are the produced and consumed active powers, $\Delta = \frac{1}{12}\,\text{h}$ converts power to energy, p_t is the cost of the most expensive generator producing, and $P^{\text{redisp}}_{g,t}$ is the deviation applied to generator g by the agent's requests and the environment's automatic balancing. The curtailment term prices the change in total curtailed renewable power P^{curt}_t and refunds released curtailment, and the official storage term is identically zero here. A blackout does not end the accounting: following the official implementation, every remaining step is charged the entire lost demand D_t, taken from the reference run and priced at $\bar{p} = \max_g p_g$,

$$C \;=\; \sum_{t=1}^{H_{\text{alive}}} c_t \;+\; \bar{p} \sum_{t=H_{\text{alive}}+1}^{H} D_t. \tag{13}$$

The episode score maps C piecewise-linearly, clipping at both ends, through four anchor costs on the same scenario: an idealized loss-minimizing operation, the official no-overflow "safe finish" reference, Do-Nothing, and an immediate whole-scenario blackout with $C_{\text{worst}} = \bar{p}\sum_{t=1}^{H} D_t$. The score F averages it over the 16 held-out scenarios,

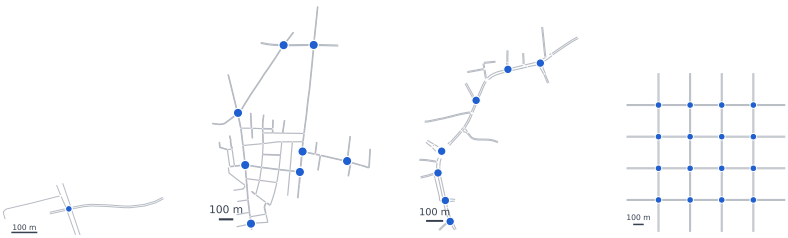
$$S = \sigma(C), \quad \sigma\colon \big(C_{\text{best}},\, C_{\text{reco}},\, C_{\text{DN}},\, C_{\text{worst}}\big) \mapsto \big(100,\, 80,\, 0,\, -100\big), \qquad F = \frac{1}{16}\sum_{s=1}^{16} S_s. \tag{14}$$

Completing a scenario more cheaply than the safe-finish reference therefore scores above 80, and operating longer and cheaper than Do-Nothing scores above 0. The task description below follows Grid2Op (Donnot, 2020) and the L2RPN competition (Marot et al., 2020).

Case14 task description

Keep every load supplied for the entire scenario. Avoid unnecessary switching, intervene only to relieve persistent overloads or restore connectivity, prioritizing blackout prevention before operating cost. Per-scenario operation cost, interpolated linearly between the best, safe-finish, do-nothing and worst references to 100, 80, 0 and -100, then averaged over the 16 held-out scenarios.

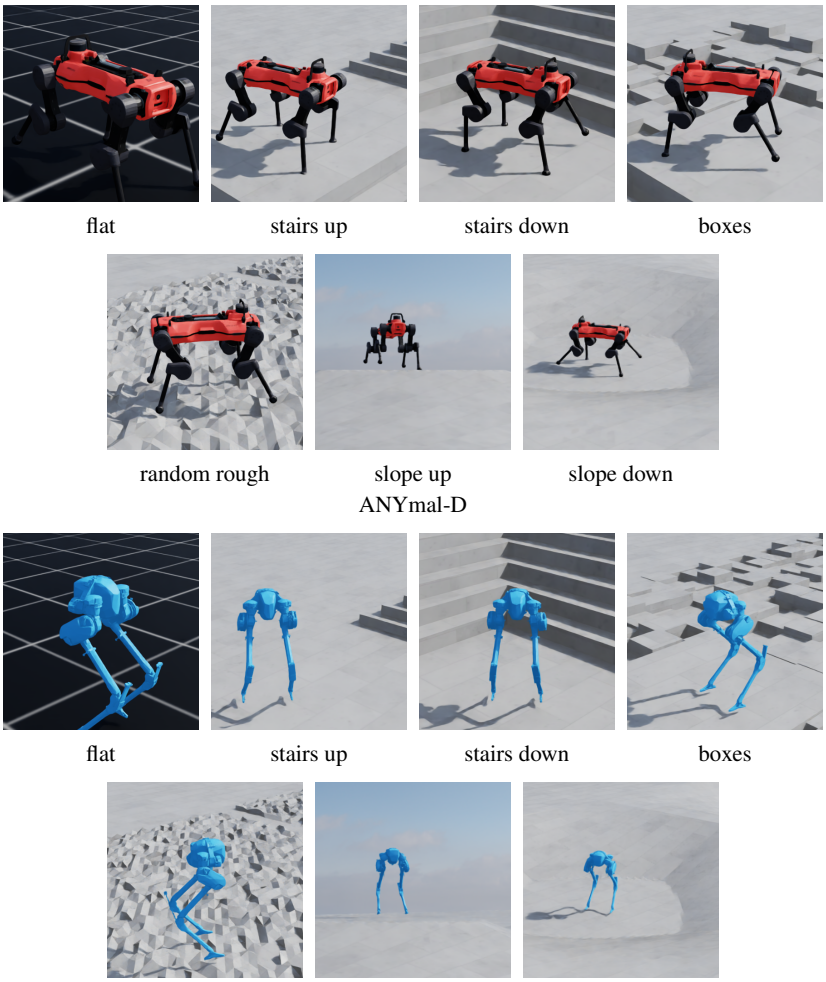
RESCO. RESCO (Ault & Sharon, 2021) couples the SUMO traffic simulator (Lopez et al., 2018) to real road networks and measured demand, and gives every signalized intersection its own learner that reads only its own entry of the designed reward. Because the learners do not communicate at execution time, the designed reward function must provide the incentives for coordination. The score $F = -\min_{e \le 40} \bar{d}_e$ is the negated average delay of the best training episode. The four networks of Figure 10 are a single real intersection, a real eight-signal corridor, an Ingolstadt district of 7 controlled signals, and a synthetic 4×4 grid. Their (obs, action) dimensions are (40, 4), (165, 23), (290, 21) and (960, 128). RESCO provides no task descriptions, so we write one per network. Each states the network and its demand, the 10 s control step with a 3 s yellow before every phase change, that unserved vehicles stay queued in the network, and the goal of minimizing vehicle delay.



Cologne1 Cologne8 Ingolstadt7 Grid4x4

Figure 10: RESCO traffic networks, with controlled signals in blue.

Isaac Lab locomotion. Isaac Lab (Mittal et al., 2025) asks the ANYmal-D quadruped and the Cassie biped to hold a commanded velocity on flat and on rough terrain, cutting the episode short on any fall, and with no binary success the score is the tracking error plus a fall charge, which bills the commanded speed for every step a fall removes from the episode. The four tasks are illustrated in Figure 11, and their descriptions follow Isaac Lab (Mittal et al., 2025).



flat stairs up stairs down boxes

random rough slope up slope down

ANYmal-D

flat stairs up stairs down boxes

random rough slope up slope down

Cassie

Figure 11: Isaac Lab locomotion environments.

B TRAINING DETAILS

Policy training. For each candidate reward, a policy is trained from scratch using the benchmark's published configuration. Only the reward function varies. Table 2 lists the training configurations for the four oracles. All six methods search under training seed 0, enforced on the server side. The selected best rewards are additionally trained with seeds 1–5 for the curves in Appendix D.

Table 2: Policy-training configuration of the four oracles.

	Algo.	Policy network	Learn. rate	Budget per candidate	Sim. step
Bi-DexHands	PPO	MLP 512−256−128	3×10^{-4}	3000 iterations	1/60 s
Grid2Op Case14	PPO	MLP 256−256	3×10^{-4}	10,240 timesteps	5 min
Isaac Lab locomotion	PPO	MLP 128−128−128 (flat) to 512−256−128 (rough)	1×10^{-3}	800 iterations	0.005 s
RESCO	DQN	Conv 2×2×64, MLP 64−64	1×10^{-3}	40 episodes	10 s

Search configuration. Table 3 lists ARBO's search configuration for the main comparisons. The schedule holds m_{explore} for the first n_0 rounds, extended until an incumbent exists, exploits thereafter, and inserts a re-exploration burst of at most ν rounds once μ consecutive rounds go unaccepted. Each attempted candidate counts as one sample.

Table 3: ARBO search configuration for the main comparisons.

Parameter	Value
Locomotion samples	72
Other domains' samples	128
Initial explore rounds, n_0	5
Exploit patience, μ	20
Re-exploration burst, ν	5

Backbones. Table 4 lists the five backbones of Figure 7c, chosen to vary the model family, the reasoning capability and the serving scale, each driven through the same OpenAI-compatible interface at the same temperature, instruction file, tools and schedule.

Table 4: Backbone ablation configurations.

Backbone	Serving	Reasoning
MiniMax-M2.7	hosted API	yes
MiniMax-Text-01	hosted API	no
DeepSeek-V4-Flash	hosted API	yes
Qwen3.8-Max	hosted API	yes
Qwen3-32B	local 4-bit quantization	yes
Temperature	0.7 for every backbone	

Experiment environment. Agent, oracle and every training server share the single node of Table 5, where the Isaac Gym and Isaac Lab oracles train one batch in parallel on the GPUs while the Grid2Op and RESCO oracles train on the CPU threads.

Table 5: Hardware and software configuration.

Environment	Specification
GPUs	4× NVIDIA GeForce RTX 4090, 48 GiB memory each
CPU	Intel Xeon Platinum 8173M, 112 threads
Memory	503 GiB
OS	Ubuntu 22.04.5 LTS
Software	NVIDIA driver 590.48.01, CUDA 13.1

Per-candidate training cost. Table 6 reports per-candidate training and evaluation times in minutes, with the mean and 10th–90th percentile range for each task.

Table 6: Per-candidate wall clock in minutes, mean and 10th–90th percentile range.

Task	Mean	Range
Bi-DexHands		
SwingCup	44	11–62
LiftUnderarm	73	56–91
DoorCloseOutward	65	57–73
Over	42	38–47
GraspAndPlace	23	16–32
BlockStack	60	51–73
BottleCap	70	65–77
Kettle	134	111–162
CatchUnderarm	46	33–62
CatchAbreast	42	31–58

Task	Mean	Range
Grid2Op		
Case14	11	10–13
Locomotion		
ANYmal-D Flat	72	48–96
ANYmal-D Rough	114	87–139
Cassie Flat	67	37–106
Cassie Rough	76	57–97

Task	Mean	Range
RESCO		
Cologne1	9	7–10
Cologne8	22	19–24
Ingolstadt7	28	26–31
Grid4x4	61	52–72

LLM inference cost. Table 7 reports each method's prefill and decode tokens per attempted candidate.

Table 7: LLM tokens per attempted candidate, averaged over the ten Bi-DexHands tasks.

Method	Prefill Tokens	Decode Tokens
Ours	**176,961**	**21,671**
RF-Agent	10,783	9,990
REvolve	3,940	3,488
Eureka	3,919	3,511
R⋆	3,391	6,132
ShinkaEvolve	19,242	7,726

C Baseline Details

Every baseline runs its own search from its released code, or from our reimplementation for R⋆, against the oracle at identical settings, with reasoning traces stripped before each method's parser sees the reply. Table 8 summarizes each method's selection and expansion rule.

Table 8: Selection and expansion rules of the compared methods.

Method	Selection	Expansion
Eureka (Ma et al., 2024c)	Previous batch's best, one per prompt.	One fixed improvement template.
REvolve (Hazra et al., 2025)	Rank-weighted from an elite archive capped at 13, one or two per prompt.	Mutation or crossover at equal rates.
RF-Agent (Gao et al., 2026)	UCT over the tree, at most four nodes per prompt.	Fixed menu of five actions.
R⋆ (Li et al., 2025)	Global best for reflection, two rank-sampled parents for crossover.	Reflection hint from {improve, remove, add} or module crossover, plus parameter alignment.
ShinkaEvolve (Lange et al., 2026)	Score-weighted parent, one archive and one top-k inspiration, at most three.	Patch type from {diff, full, cross} at fixed rates.
Ours	Chosen by the agent with tools over the full history, failures included.	Free-form edits from its diagnosis, batch size its own.

Eureka. We set the per-iteration batch size to 4.

REvolve. We use the fully automated variant without the human-feedback loop, and set 4 candidates per iteration with the elite archive capped at 13.

RF-Agent. We set the per-expansion action counts to $(2, 2, 2, 1, 1)$ over its five actions, eight children per expansion, and the UCT constant annealed linearly from 0.4 to 0.1, with thought-align and self-verify enabled.

R⋆. In the absence of an official implementation, we reimplement the method from the paper. We use a population of 16, generate 12 reflection offspring and 4 module-level crossover offspring per iteration, and employ 5 LLM critics for Bradley-Terry parameter alignment. We retrain the top 2 individuals at each iteration.

ShinkaEvolve. We set one candidate per generation, 2 islands, an archive of 40 with elite ratio 0.3, weighted parent selection at $\lambda = 10$, one archive and one top-k inspiration per prompt, and diff, full and cross patch probabilities of 0.6, 0.3 and 0.1, with the LLM novelty judge on, while the embedding-based novelty filter is disabled.

D Additional Results

D.1 Dexterous Manipulation

Figure 12 shows per-task best-so-far scores and training curves. The upper two rows plot the best-so-far success against reward-function samples on the ten Bi-DexHands tasks. The lower two rows plot the success rate of the policy trained with each method's best reward against environment steps. Every method is given the same budget of 128 reward-function samples.

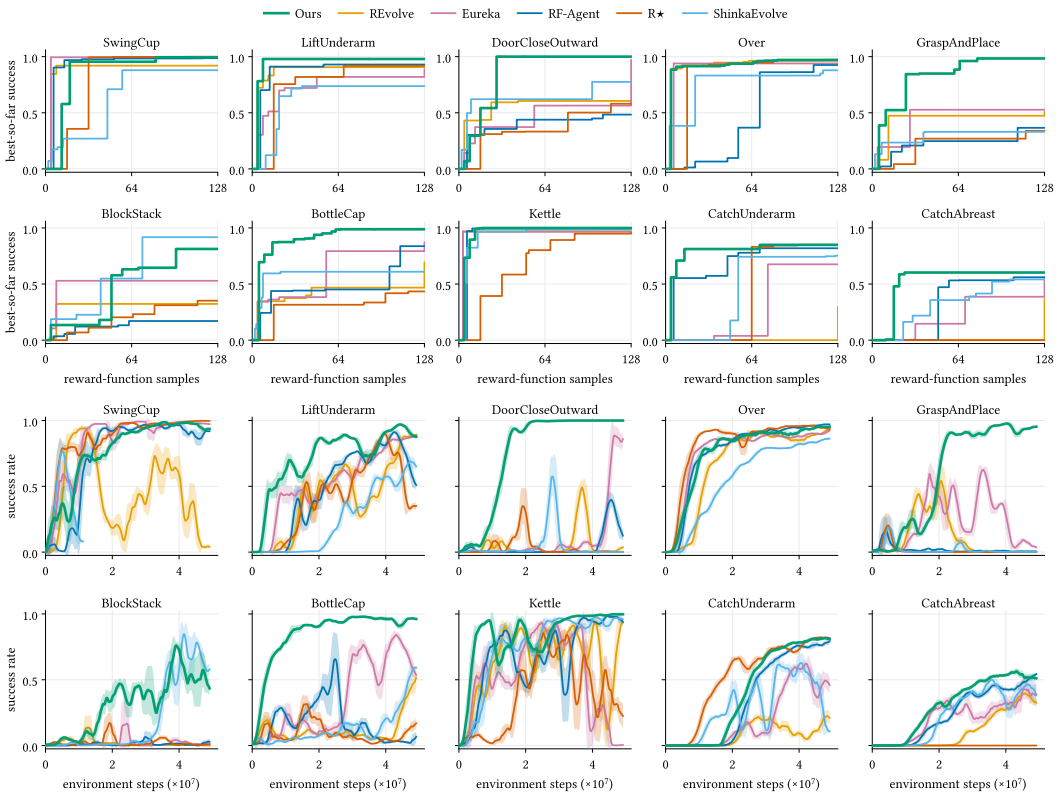


Figure 12: Bi-DexHands search and training curves.

D.2 Power-Grid Operation

Per-scenario survival. Figure 13 replays the official evaluation episodes of each method's best designed reward. Ours completes the most four-week scenarios, ShinkaEvolve completes a few, and no other method completes any. A survival fraction of 1.0 denotes completion of the full scenario. Reward design has to turn survival into a per-step signal on the loading margin. The control task is sparse, since a scenario is scored only at a blackout or at its end.

Line-loading trajectories. Figure 14 shows the maximum line loading ρ during each episode. Each panel is one of the sixteen evaluation scenarios, replayed under each method's best designed reward

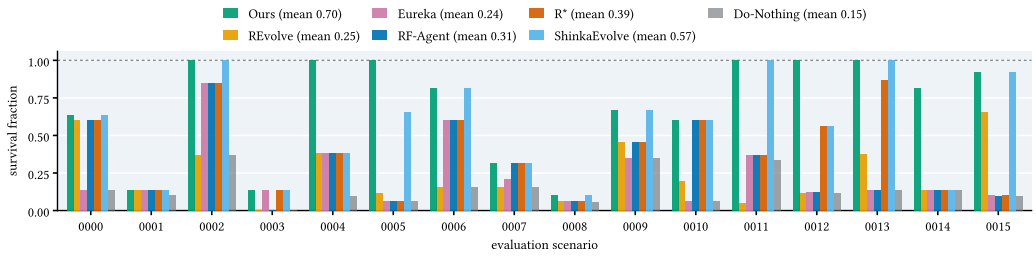


Figure 13: Case14 per-scenario survival.

over four weeks. Dashed curves denote Do-Nothing, and crosses mark blackouts. The red line marks the thermal limit at $\rho = 1$, and the legend lists the official score of each method's best reward.



Figure 14: Case14 line-loading trajectories.

D.3 Locomotion

Figure 15 shows the best-so-far scores during reward search, with performance under the environment reward as a reference, and the PPO training curves for each method's best reward. The panels cover ANYmal-D Flat, ANYmal-D Rough, Cassie Flat, and Cassie Rough, with 72 reward-function samples per method and 800 PPO iterations per policy. Search scores are negated fall-charged tracking errors, and the training panels plot the fall-charged tracking error and the fall rate.

D.4 Traffic-Signal Control

Figure 16 presents the best-so-far scores as a function of the number of evaluated reward candidates and the policy training curves for the reward selected by each method. The panels cover Cologne1, Cologne8, Ingolstadt7, and Grid4x4, with the environment reward as a reference in the search panels.

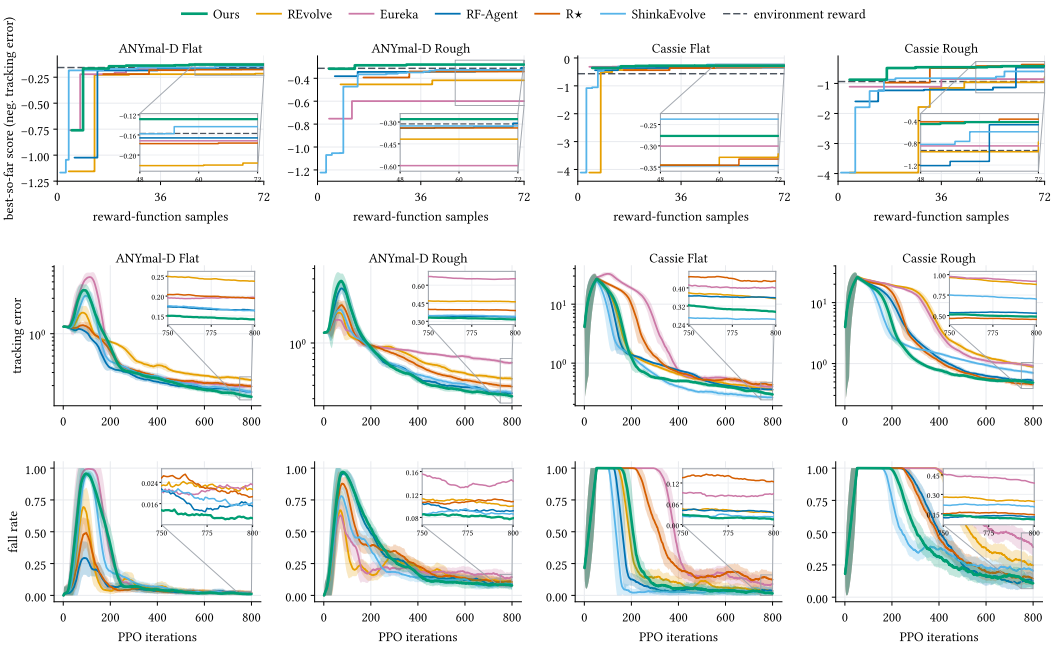


Figure 15: Locomotion search and training curves.

Each method searches for 128 reward-function samples, and each policy trains for 40 episodes. The training panels plot the average delay per training episode as the median over five held-out seeds. Bands show the interquartile range across seeds, and insets enlarge the last 10 episodes.

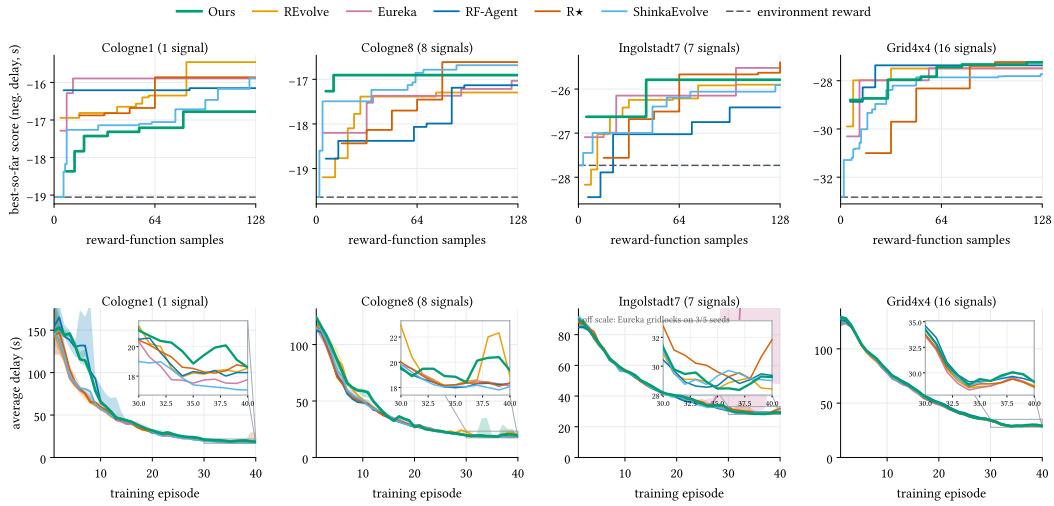


Figure 16: RESCO search and training curves.

D.5 Ablations

Search curves. Figure 17 plots the trajectories the bars of Figure 7 end on, each bar the loss as a share of the full method's gain over the environment reward, Do-Nothing on Case14, or Sparse on GraspAndPlace. The four panels show the best-so-far search score against reward-function samples on Cassie Flat, Cologne8, Case14, and GraspAndPlace.

Training curves. Figure 18 shows the training curves behind the Cassie Flat, Cologne8, and GraspAndPlace curves of Figure 17 through the environment's own diagnostics. Figure 18a and Figure 18b plot the fall-charged tracking error and the fall rate on Cassie Flat, Figure 18c the average delay on Cologne8, and Figure 18d the success rate on GraspAndPlace.

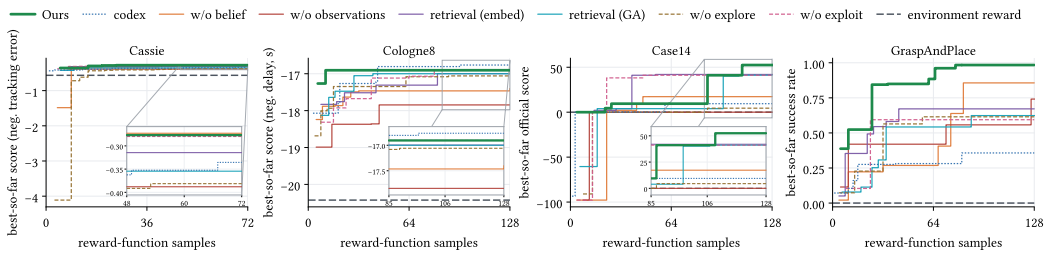


Figure 17: History and schedule ablation search curves.

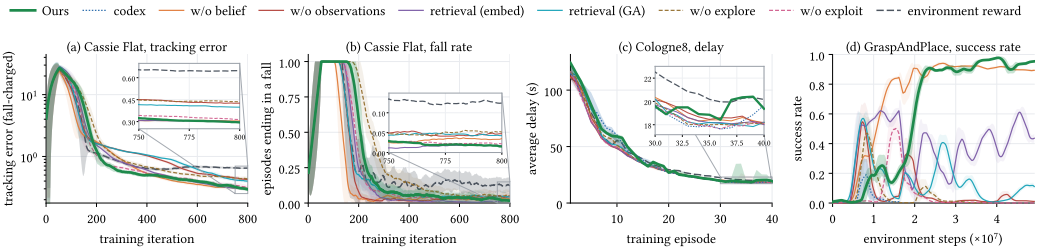


Figure 18: History and schedule ablation training curves.

Run on different backbones. Figure 19 presents the best-so-far search scores and policy training curves for each backbone's best reward, using the backbones of Figure 7c, whose axes run from the reference reward at 0 to the best backbone at 1. MiniMax-Text-01 and Qwen3-32B perform poorly, particularly on manipulation. The method thus needs a capable backbone.

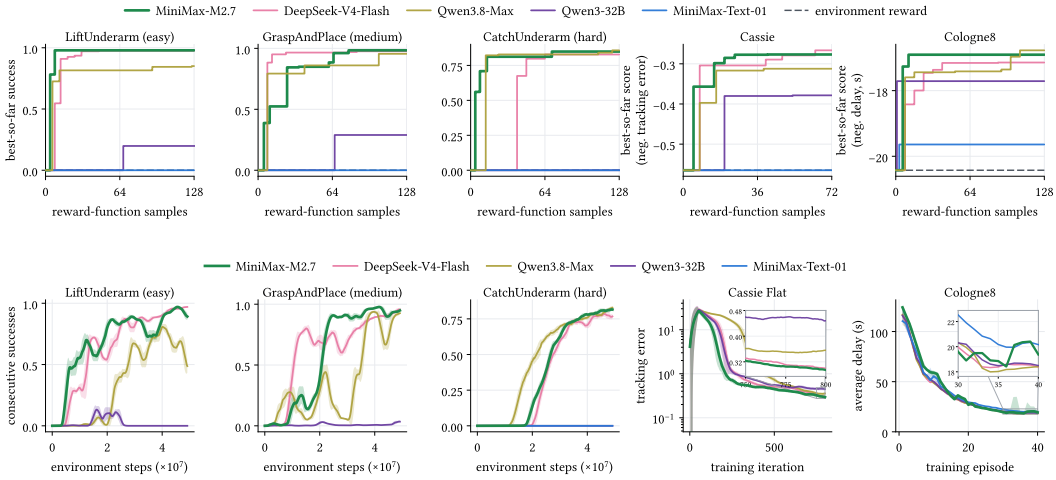


Figure 19: Backbone ablation search and training curves.

E ALGORITHMS

Algorithm 1 gives the full procedure of the search described in Section 4.

Algorithm 1 ARBO: Agentic Reward Black-box Optimization.

Require: task specification $\mathcal{T}$, sample budget B, schedule constants n_0, μ, ν

$t \leftarrow 1$; $\mathcal{H}_1 \leftarrow \varnothing$; $\mathcal{B}_1 \leftarrow \varnothing$; $\hat{R}_0 \leftarrow \varnothing$; $F(\varnothing) \leftarrow -\infty$
while the sample budget B is not spent **do**
  $n_{\text{rej}} \leftarrow$ consecutive rounds with $a_i = 0$ **and** $m_i = m_{\text{exploit}}$ before t
  $n_{\text{exp}} \leftarrow$ consecutive rounds after round n_0 with $m_i = m_{\text{explore}}$ before t
  if $t \le n_0$ **or** $\hat{R}_{t-1} = \varnothing$ **then** ▷ *schedule* ψ
    $m_t \leftarrow m_{\text{explore}}$
  else if $n_{\text{rej}} \ge \mu$ **or** $\big(0 < n_{\text{exp}} < \nu$ **and** $a_{t-1} = 0\big)$ **then**
    $m_t \leftarrow m_{\text{explore}}$
  else
    $m_t \leftarrow m_{\text{exploit}}$
  end if
  $\mathcal{D}_t \leftarrow \mathcal{H}_t \cup \mathcal{B}_t$ ▷ *history*
  $\{R_t^k\}_{k=1}^{N_t} \sim g\big(\cdot \mid \mathcal{T}, \mathcal{D}_t, m_t\big)$ ▷ *proposal operator*
  $\mathcal{B}_{t+1} \leftarrow G_{t+1} \cup \Lambda_{t+1}$
  for $k = 1, \dots, N_t$ **do**
    $(y_t^k, \ell_t^k, e_t^k) \leftarrow \mathcal{O}(R_t^k)$ ▷ *evaluation oracle*
  end for
  $\mathcal{H}_{t+1} \leftarrow \mathcal{H}_t \cup \{(R_t^k, y_t^k, \ell_t^k, e_t^k)\}_{k=1}^{N_t}$
  $k^\star \leftarrow \arg\max_k y_t^k$; $a_t \leftarrow \mathbf{1}\big[y_t^{k^\star} > F(\hat{R}_{t-1})\big]$
  $\hat{R}_t \leftarrow R_t^{k^\star}$ **if** $a_t = 1$ **else** $\hat{R}_{t-1}$
  $t \leftarrow t + 1$
end while
return the best reward $\hat{R}_{t-1}$, the highest-scoring candidate in $\mathcal{H}_t$

F LICENSES

Table 9 lists the licenses and URLs of the baselines, benchmarks and training algorithms we use in this work. R⋆ releases no code, and our reimplementation follows Appendix C.

Table 9: Resource licenses and URLs.

Resources	License	URL
Eureka (Ma et al., 2024c)	MIT	https://github.com/eureka-research/Eureka
REvolve (Hazra et al., 2025)	Available online	https://github.com/RishiHazra/Revolve
RF-Agent (Gao et al., 2026)	Available online	https://github.com/deng-ai-lab/RF-Agent
ShinkaEvolve (Lange et al., 2026)	Apache 2.0	https://github.com/SakanaAI/ShinkaEvolve
Isaac Gym (Makoviychuk et al., 2021)	BSD 3-Clause	https://github.com/isaac-sim/IsaacGymEnvs
Bi-DexHands (Chen et al., 2023)	Apache 2.0	https://github.com/PKU-MARL/DexterousHands
Grid2Op (Donnot, 2020)	MPL 2.0	https://github.com/Grid2Op/grid2op
Isaac Lab (Mittal et al., 2025)	BSD 3-Clause	https://github.com/isaac-sim/IsaacLab
RESCO (Ault & Sharon, 2021)	CC BY-NC-SA 4.0	https://github.com/Pi-Star-Lab/RESCO
SUMO (Lopez et al., 2018)	EPL 2.0	https://github.com/eclipse-sumo/sumo
rl-games (Makoviichuk & Makoviychuk, 2021)	MIT	https://github.com/Denys88/rl_games
Stable-Baselines3 (Raffin et al., 2021)	MIT	https://github.com/DLR-RM/stable-baselines3
RSL-RL (Rudin et al., 2022)	BSD 3-Clause	https://github.com/leggedrobotics/rsl_rl
PFRL	MIT	https://github.com/pfnet/pfrl

G PROMPTS

The fixed context c_t of Equation (8) consists of the texts shown below. The **instruction file** is the fixed instruction of the proposal operator. The **search mode guidance** is the steering prompt m_t of Equation (11), and its two values differ only in the mode named beneath the protocol text. The **task prompt** carries the task specification $\mathcal{T}$, the task description whose source Appendix A cites, with its domain reward contract and the acceptance metric F.

Instruction file. The instruction file states what the workspace holds and what one round must produce, and it leaves the task specification to the slots highlighted in blue.

Instruction file: the per-round workflow.

```
---
name: reward_design
scope: subagent
agent: reward_design
description: >-
  Design and improve the training reward for a reinforcement-learning task. Read the
  staged task and the history of past attempts, write this attempt's candidate rewards,
  and commit them once for evaluation.
initial_message: >-
  Improve the training reward for this reinforcement-learning task:

  {{ task_specific_prompt }}

  The task prompt above is authoritative for this task, and it names the reward contract,
  the files you write, and the acceptance metric the loop maximizes.

  `.arbo_history/observations.jsonl` in the staged code package is the record of every
  past attempt and of every candidate inside it, and `analysis.md` holds your own notes
  from the last one. An absent or empty history means this is the first attempt.

  You have {{ max_turns }} tool-calling turns, and the attempt scores nothing until
  candidates are written and committed, so make your first write by turn {{ write_by_turn
  }}.

  Each attempt you read what you need, write this round's candidates, overwrite `
  analysis.md` with your evidence and one hypothesis per candidate, and commit them in
  exactly one commit. Then stop and let the host evaluate.

  {{ search_mode_guidance }}

  search_mode: {{ search_mode }}
---

# Reward design

Your reward is a training signal. The acceptance metric is fixed, it is defined by the
  task, and it is never yours to edit or reimplement. Your reward rising while the
  acceptance metric does not is the failure this loop selects against.

The task file in the staged code package, named in the task prompt, is the complete task.
  It holds the state your reward may consume, with names, shapes, frames and units, and it
  holds the code that computes the acceptance metric. Nothing outside the files the task
  prompt names is yours to edit.

`observations.jsonl` records every candidate and not only the winners, so each one carries
  its score, its per-term diagnostics, and its traceback if it crashed. A crashed
  candidate still spent a sample and says nothing about the hypothesis behind it.

Training is deterministic, and a candidate whose code repeats an earlier one is not
  retrained but still costs a sample.
```

Search mode guidance. The search mode guidance is the static explore/exploit protocol, injected into the context each round by the schedule.

Search mode guidance: explore/exploit protocol.

```
The host sets your mode each attempt and scores the result, and it never runs git, so you
  run every git command yourself inside the staged `code_package`. Every attempt ends with
  exactly one commit on a named branch, never on a detached HEAD, where `main` carries the
  incumbent best line and each explore probe lives on its own `explore/*` branch.
```

```
In exploit mode you refine the current best, keeping the method and tuning it rather than
  discarding what works, so run `git checkout main` first, and if your last winning probe
  is still on an `explore/*` branch, fold it in with `git merge --no-ff <branch>` before
  you commit. In explore mode you propose a structurally different candidate, a different
  method rather than different constants, so run `git checkout -b explore/<short-name>
  <bootstrap-sha>` and branch off the bootstrap root rather than `main` or an earlier
  probe. Either way the host accepts or rejects by score alone, independent of which
  branch you chose.
```

Task prompt. The task prompt is rendered from the task adapter when the workspace is staged. The text below shows its slots in double braces.

Task prompt: the per-task slot.

```
Task id: {{ task_id }}
{{ task_description }}

Write N candidate rewards this attempt, N in [1, {{ N_max }}]; files beyond the cap are
  dropped. The host trains all N in parallel and every file costs one sample. This run
  must reach {{ max_samples }} evaluated samples within at most {{ max_attempts }}
  attempts; observations.jsonl shows how many of each are used.
Put them in this attempt's candidate folder, named in your instructions, as reward1.py ..
  rewardN.py together with this attempt's analysis.md, and commit them in one commit. Only
  that folder is trained; earlier iteration_* folders are read-only history.

Reward contract: {{ reward_contract }}

shared/input_assets/code_package/env.py is the full composed task and is read-only. Every
  candidate trains for the same fixed budget.

The loop steers explore/exploit by the best candidate's `{{ acceptance_key }}` -- this
  task's acceptance metric ({{ metric_kind }}; {{ metric_range }}). The task description
  above states the control objective; read the top-level env.py documentation and the code
  that computes `{{ acceptance_key }}` for its exact evaluation formula. Design each
  candidate as a TRAINING reward that teaches the policy to achieve that objective; never
  replace or imitate the fixed evaluator mechanically. The metric may be negative or
  unbounded. Raise it (higher is always better), and do not assume a [0,1] range.

Metrics in observations.jsonl: `{{ acceptance_key }}_*` is the acceptance metric ({{
  metric_kind }}; {{ metric_range }}). The reported score is the {{ aggregation }} of `{{
  acceptance_key }}`; the loop accepts an attempt ONLY if it rises. Judge by TREND, not
  absolute value -- do not assume a [0,1] range. `gpt_reward_*` is the reward the policy
  actually optimized (your designed reward's realized value) -- its scale/rise is not by
  itself task progress. `rew_<term>_*` are your per-term diagnostics revealing dead,
  saturated, exploding, or dominating components.

Each line of .arbo_history/observations.jsonl is one attempt: `mode`, `decision`, `reward`
  (its acceptance value, `{{ acceptance_key }}`), and `raw.candidates[]` with one record
  per candidate (`file`, `score`, `metrics`, `series`, `error`, `train_log`). The lines
  are large; project the fields you want with python or jq.
```

AutoResearch with Codex. *codex* in Figure 7b receives the prompt below.

AutoResearch with Codex: the session prompt.

```
Design the training reward for a reinforcement-learning control task.

## Task: {{ task_id }}

{{ task_description }}

## Files

`reward.py` holds the reward function. It starts as a placeholder that returns
zero everywhere and trains nothing -- a valid starting point, not a reward.

`env_reference.py` is the complete environment code, read-only. It defines the
state tensors a reward may read, the episode structure, the resets, and the
success criterion. Read it to learn which `self.<name>` tensors exist and what
they mean.

## Reward contract

{{ reward_contract }}

## Scoring a reward
```

```
    evaluate <file.py> [<file.py> ...]

Each file is one candidate: the command trains a policy on it with {{ rl_algorithm }}
from scratch and scores the resulting policy by the task's own acceptance metric. The
metric is fixed and you cannot edit or influence it except through the reward you write:

  * `{{ acceptance_key }}` -- {{ metric_kind }}
  * range: {{ metric_range }}
  * aggregation: {{ aggregation }}

{{ official_evaluator_note, when the task has one }}

For each candidate it reports the score, the training statistics of your reward's own
named terms, the tail of the training log, and the full traceback if it failed. It is
the only way to find out how a reward performs -- there is no network here and no way
to run the environment yourself.

The command blocks: training is real and takes roughly {{ eval_minutes }} minutes per
candidate. Candidates passed in one call train in parallel, so a call with several
candidates costs about as much wall-clock as a call with one.

## Budget

You have {{ n_budget }} evaluations for this task. Every file you pass to `evaluate` spends
one, whether or not the reward turns out to work. Nothing else is measured, and there
is nothing beyond the budget: when it is spent, `evaluate` refuses.

Your goal is the highest `{{ acceptance_key }}` that any single candidate of yours
reaches within those {{ n_budget }} evaluations. How you get there is up to you.
```