R-SPLIT: POPULATION-LEVEL REASONING OVER REWARD MECHANISMS FOR AUTOMATED REWARD DESIGN WITH LLMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Reward functions strongly affect how reinforcement learning (RL) agents learn, but designing them for complex control tasks requires substantial expert effort. Large language models (LLMs) can generate and improve reward code. However, existing methods typically apply LLM reasoning to individual candidates, which limits their use of population-wide information for candidate comparison and search planning. We propose R-Split, which analyzes candidate reward designs, compares them across the population, makes evolution decisions, and generates new reward code. R-Split summarizes each evaluated reward function as a structured set of **reward mechanisms**: textual descriptions of how its components shape learning, grounded in reward code and training-process data. A similarity filter removes candidates with highly similar mechanism descriptions, reducing repeated exploration. We evaluated our approach against multiple baselines on ten low-level control tasks, using a shared PPO evaluation pipeline and five independent runs for each selected reward. R-Split achieves competitive performance across both locomotion and manipulation tasks. These results show that comparing reward mechanisms across a population can help improve LLM-based reward design.

1 INTRODUCTION

Reinforcement learning (RL) has achieved remarkable success in low-level control tasks such as dexterous manipulation, legged locomotion, and autonomous flight. If an agent receives reward only upon task completion, the resulting feedback is sparse and delayed, making effective policy learning difficult. Reward shaping addresses this problem by supplying dense, incremental signals that reflect intermediate task progress (Ng et al., 1999). Traditionally, such shaping rewards have been manually designed by domain experts, requiring substantial task knowledge and potentially inducing unintended behavior when poorly specified (Booth et al., 2023). Another line of work learns rewards from data: inverse RL infers reward functions from demonstrations, whereas preference-based methods learn reward models from trajectory preferences (Ziebart et al., 2008; Christiano et al., 2017). These methods reduce the need for direct manual reward design but require demonstration data or external feedback.

Large Language Models (LLMs), with strong code generation and reasoning capabilities as well as broad domain knowledge, have enabled automated reward design. Existing methods use LLMs to generate, revise, or recombine reward functions, but predefined optimizers still control parent selection and next-generation decisions (Ma et al., 2024; Hazra et al., 2025; Gao et al., 2025; Li et al., 2025a; Sun et al., 2025). This leaves the reasoning capability of LLMs underused in population search. We propose **R-Split**, which applies LLM reasoning to all candidates in a population to make more informed parent-selection and generation decisions, thereby making better use of a limited reward-evaluation budget.

Directly comparing raw reward code across a population is difficult because implementation syntax and numerical parameters can obscure the underlying design ideas and make semantically similar designs appear different. R-Split therefore represents every evaluated candidate as a structured record of reward mechanisms grounded in training-process data. A reward mechanism describes

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.

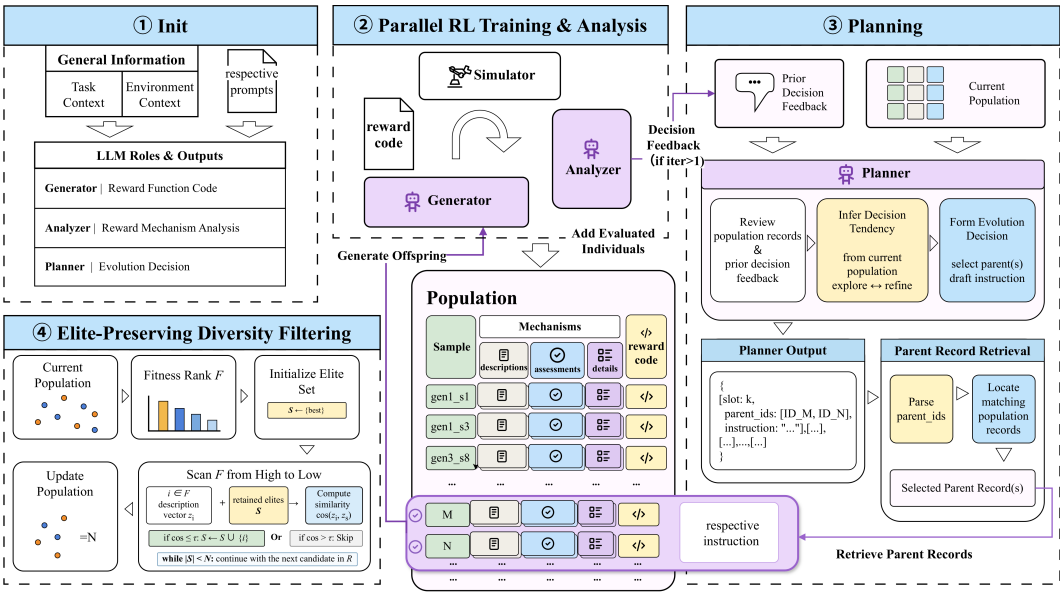

Figure 1: Overview of the R-Split workflow.

how one or more reward components transform and combine task signals to shape learning. Each record contains a description of the reward mechanism, an assessment based on training-process data, and implementation details grounded in the code, so that the LLM can use a common representation to select parents from the population. Natural-language sketches have also been found effective as intermediate representations for LLM code generation (Deng et al., 2024), and EoH (Liu et al., 2024) similarly connects high-level heuristic descriptions with executable code.

R-Split first generates multiple reward candidates and trains and evaluates their policies in parallel, testing several reward-design directions early in the search. It then extracts mechanism records from their reward code and training-process data. By comparing these records across the population, R-Split identifies promising directions, determines which designs to retain or refine, whether mechanisms can complement each other, and when to explore new directions. It then selects parents, makes evolution decisions, and uses those decisions to generate the next generation of reward programs.

In summary, our contributions are twofold: (1) we enable LLMs to compare reward mechanisms across an entire population and jointly plan the next generation through structured mechanism records; and (2) we systematically evaluate R-Split on ten low-level control tasks and show that it performs competitively.

2 Related Work

2.1 Reward Design with Large Language Models

Some approaches use a foundation model itself as an online proxy reward model. An LLM can be prompted with examples or a description of desired behavior and queried during training to evaluate agent behavior and return scalar rewards (Kwon et al., 2023). FuRL instead uses a pretrained VLM to provide fuzzy rewards for sparse-reward visual control and fine-tunes its representation to reduce reward misalignment (Fu et al., 2024). In contrast, Language to Rewards (L2R) and Text2Reward use LLMs to synthesize executable reward programs from language instructions and structured environment interfaces; the generated programs, rather than the LLMs, then compute rewards (Yu et al., 2023; Xie et al., 2024).

Subsequent work differs mainly in how it organizes search and uses feedback. Eureka greedily refines the best candidate from the previous generation (Ma et al., 2024), whereas REvolve maintains an evolutionary population and incorporates human feedback (Hazra et al., 2025). RF-Agent formu-

lates reward design as sequential decision making and organizes search histories with Monte Carlo Tree Search (Gao et al., 2025). R* separates reward structure evolution from parameter alignment optimization (Li et al., 2025a). LaRes couples reward search with shared replay, reward relabeling, and Thompson-sampling-based policy selection to improve interaction efficiency (Li et al., 2025b). CARD uses process, trajectory, and trajectory-preference feedback to revise reward code without human feedback (Sun et al., 2025). RDA uses vision-language models to diagnose behavioral failure modes from trajectories and revise reward code accordingly (Lee et al., 2026). CRWM learns causal relationships between reward components and physical variables from offline multi-task data, then uses this prior to constrain zero-shot reward generation without task-specific refinement (Yang et al., 2026). Collectively, these approaches span direct reward generation, evolutionary and tree-based search, structure–parameter decomposition, multimodal feedback, and feedback-free reward synthesis.

2.2 Code LLMs for Automated Design

Code-trained LLMs can synthesize executable programs from language specifications, including robot policy code from natural-language commands (Liang et al., 2023). Research on code LLMs increasingly separates high-level planning from implementation. Self-planning code generation first elicits a sequence of solution steps and then generates code under that plan (Jiang et al., 2024). MapCoder decomposes competitive program synthesis into example retrieval, planning, code generation, and debugging roles (Islam et al., 2024). A systematic comparison of natural-language solutions, pseudocode, and programming languages as intermediate representations finds natural language to be the most consistently effective representation in the evaluated settings (Deng et al., 2024).

Beyond solving specified programming tasks, code LLMs have also been used for open-ended program and algorithm search. FunSearch combines a pretrained code model with an automated evaluator and a program database to iteratively generate and select executable functions (Romera-Paredes et al., 2024). EoH jointly evolves natural-language heuristic ideas and their executable implementations (Liu et al., 2024). ReEvo instead combines evolutionary search with LLM-generated reflections that act as verbal guidance for subsequent heuristic generation (Ye et al., 2024). Together, these studies show that code LLMs can use planning, intermediate descriptions, reflective feedback, and executable programs to assist automated design.

3 Problem Setting

We search for an executable reward program c for a given control task. We evaluate c using the area under its task-score curve (AUC) during policy training. AUC favors reward programs that support both early learning and sustained performance rather than a transient peak.

R-Split maintains a population of evaluated reward programs. Each individual i contains reward code c_i, score AUC_i, RL training-process data e_i (curves and sampled values), reward mechanism analysis d_i, and decision feedback q_i when applicable. The Generator creates the first generation from scratch. At iteration t, $\mathcal{P}^t$ denotes the current population provided to the Planner; it contains at most $2N$ individuals. The search returns the maximum-AUC reward program among all evaluated individuals.

4 Method

Figure 1 summarizes the R-Split workflow, and Algorithm 1 formalizes the search procedure. The task description and environment interface information are provided as background to all three LLM roles, together with their role-specific prompts. In each cycle, the Planner compares the population, the Generator creates rewards from selected parents, policies are trained in parallel, the Analyzer evaluates their mechanisms, and the diversity filter retains high-AUC candidates while reducing repeated mechanism descriptions.

Each RL training operation trains a policy with one reward program and collects its training-process data. Operations marked parallel are executed concurrently. The search returns the reward program with the highest AUC among all evaluated individuals.

Algorithm 1: R-Split reward-design search

Require: Task description, environment interface, population size N, total iterations T

$t \leftarrow 1$
parallel: $c_k^1 \leftarrow \text{Generator}(\emptyset, \emptyset)$, $k = 1{:}N$
parallel: obtain i_k^1 by RL training with c_k^1, $k = 1{:}N$
$\mathcal{P}^1 \leftarrow \{i_k^1\}_{k=1}^N$
for $t = 2, \ldots, T$ **do**
  parallel: $(d_k^{t-1}, q_k^{t-1}) \leftarrow \text{Analyzer}(i_k^{t-1})$, $k = 1{:}N$
  $\{(f_k^{t-1}, u_k^{t-1})\}_{k=1}^N \leftarrow \text{Planner}\big(\{(\text{AUC}_i, d_i^{\text{P}}, q_i)\}_{i \in \mathcal{P}^{t-1}}\big)$
  parallel, for $k = 1{:}N$:
  $c_k^t \leftarrow \text{Generator}\big(f_k^{t-1}, \{d_i^{\text{G}} \mid i \in f_k^{t-1}\}, u_k^{t-1}\big)$
  $\mathcal{P}^t \leftarrow \text{Select}(\mathcal{P}^{t-1}, N)$
  parallel: obtain i_k^t by RL training with c_k^t, $k = 1{:}N$
  add $\{i_k^t\}_{k=1}^N$ to $\mathcal{P}^t$
end for
return $\arg\max_i \text{AUC}_i$

4.1 Mechanism Analysis

The Analyzer receives reward code c_i and training-process data e_i. For an individual produced by an evolution decision, h_i additionally records its selected parents and instruction. The Analyzer outputs a reward mechanism analysis d_i and decision feedback q_i:

$$(d_i, q_i) = \text{Analyzer}(c_i, e_i, h_i), \tag{1}$$

where h_i and q_i are empty when no preceding evolution decision exists. Each nonempty q_i is paired with the corresponding individual in the Planner input.

A reward component is an individual term in the code, whereas a reward mechanism describes how one or more components use, transform, and combine task signals to shape learning. Its description retains the signal source, transformation, relationship, and functional purpose without fixing numerical values, enabling comparison across individuals.

The analysis is structured as

$$\begin{aligned} d_i &= \{\text{description}, \text{assessment}, \text{detail}\}_i, \\ d_i^{\text{P}} &= \{\text{description}, \text{assessment}\}_i, \\ d_i^{\text{G}} &= \{\text{description}, \text{detail}\}_i. \end{aligned} \tag{2}$$

The description identifies each reward mechanism. The assessment evaluates it using training-process data. The detail provides a numerical analysis of its implementation, including formulas, parameter values, recorded scales, and trajectories. The Generator receives d_i^{G} together with the parent code and training-process data. Decision feedback reports the performance of the offspring generated by the preceding evolution decision.

4.2 Planning and Generation

At iteration t, the Planner receives the AUC, mechanism analysis d_i^{P}, and decision feedback for each individual in $\mathcal{P}^t$. It jointly maps these records to N next-generation evolution decisions:

$$\begin{aligned} &\{(f_k^t, u_k^t)\}_{k=1}^N \\ &\quad = \text{Planner}\left(\{(\text{AUC}_i, d_i^{\text{P}}, q_i)\}_{i \in \mathcal{P}^t}\right), \end{aligned} \tag{3}$$

where $f_k^t \subseteq \mathcal{P}^t$ is the parent set selected for slot k, each $i \in f_k^t$ is a selected parent, and $|f_k^t| \in \{0, 1, 2\}$. The corresponding instruction is u_k^t.

The Planner first analyzes all candidates. When they perform poorly, it explores new designs. When strong candidates appear, it refines them or combines complementary mechanisms. For each slot,

the Planner specifies which mechanism or relationship to preserve or reconsider and what outcome to pursue; concrete reward terms, transformations, formulas, and parameters remain the Generator's responsibility.

For slot k, the Generator retrieves the complete parent record(s) selected by the Planner and produces reward code

$$c_k^{t+1} = \text{Generator}\big(f_k^t, \{d_i^{\text{G}} \mid i \in f_k^t\}, u_k^t\big)\,. \tag{4}$$

The Generator uses the parent records and mechanism analyses d_i^{G} to select reward terms and determine how to transform, combine, and scale them. It may revise local mechanisms as needed to realize the evolution decision while keeping all other parts unchanged.

After the Planner processes the population information at iteration t, elite-preserving diversity filtering begins and retains up to N individuals as the basis of $\mathcal{P}^{t+1}$. The evaluated next-generation individuals $\{i_k^{t+1}\}_{k=1}^N$ are then added before the next Planner call. Candidates are scanned in descending task-score AUC. A candidate is discarded if the cosine similarity between its mechanism descriptions and those of any retained candidate exceeds the threshold. If fewer than N candidates remain, the threshold is increased and filtering is repeated. We use the empirical initial threshold 0.80 in all experiments.

5 Experiments

5.1 Experiment Setup

We evaluate R-Split on ten low-level control tasks from Isaac Gym (Makoviychuk et al., 2021) and Bi-DexHands (Chen et al., 2022). All methods use the same task environments and PPO configurations (Schulman et al., 2017). We compare R-Split with Eureka (Ma et al., 2024), REvolve (Hazra et al., 2025), RF-Agent (Gao et al., 2025), R* (Li et al., 2025a), and CARD (Sun et al., 2025). Eureka and RF-Agent use their public implementations, and REvolve uses the feedback-free implementation distributed with RF-Agent. We reimplement R* and CARD from their published descriptions. The expert reference uses the human-designed reward functions distributed with Eureka.

R-Split, Eureka, REvolve, and R* each run for $T = 5$ iterations with $N = 16$ candidates per iteration; RF-Agent searches 80 nodes, while CARD follows its original five-iteration sequential optimization protocol. All LLM calls use GPT-5-mini. Reward search uses a fixed training seed of 42, and experiments run on two NVIDIA RTX 4090 GPUs.

Each method selects its best reward from the search history according to its original criterion; R-Split selects the candidate with the largest AUC. We then train a policy from scratch with each selected reward using seeds 0–4 and report the mean and standard deviation. Evaluation metrics include maximum task score and AUC. Because task-score scales differ across environments, values are compared only within each task. For the search-progress comparison, each candidate is summarized by its AUC divided by the training horizon, which yields its average task score over training. R-Split, Eureka, REvolve, and R* use their native 16-candidate generations. Because RF-Agent generates eight children from a fixed prompt at each node expansion, two consecutive expansions are grouped as one 16-candidate comparison iteration. CARD retains its native sequential iterations. The expert reference is based on evaluation data. With the same search budget of five generations and 16 candidates per generation, R-Split uses 18.07M API-reported tokens across ten tasks, 4.65 times Eureka's 3.88M; counts include prompt, completion, and provider-reported reasoning tokens (Appendix A.6).

5.2 Performance Evaluation

Table 1 summarizes the results for each task. R-Split remains competitive across the evaluated tasks and consistently ranks among the leading automated methods in both AUC and maximum task score. The two metrics are complementary: AUC captures early and sustained learning, whereas maximum task score captures peak performance during training. The expert reward remains a task-specific reference rather than a common upper bound.

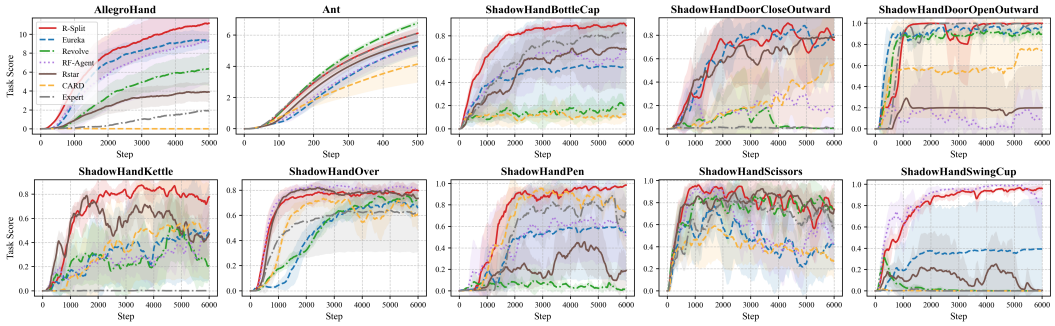

Figure 2: Task-score trajectories during reward evaluation across the ten tasks. Curves show the mean over five independent evaluation runs; shaded regions show ± 1 standard deviation.

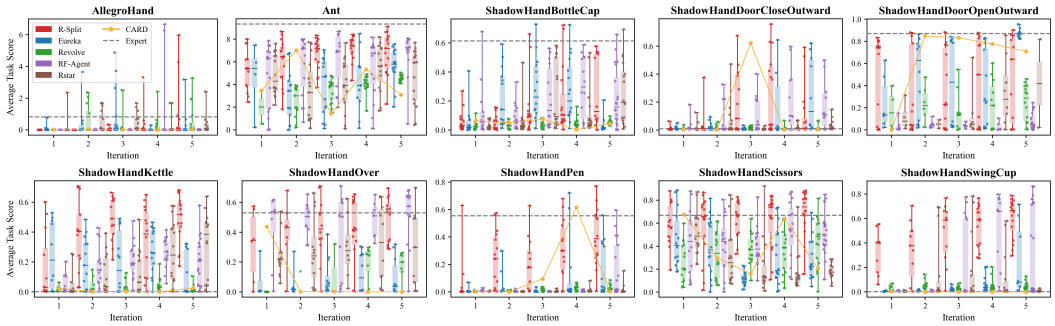

Figure 3: Search-progress comparison across the ten tasks. Each candidate is summarized by its task-score AUC divided by the training horizon. For population methods, one iteration is one 16-candidate generation. For RF-Agent, one iteration consists of two consecutive expansions, each generating eight children; CARD retains its native sequential iteration. The expert line is based on evaluation data.

Table 1: Results for each task from five independent runs. Panel (a) reports AUC in units of 10^3; values are comparable only within the same task. Panel (b) reports maximum task score. Results are mean $\pm$ standard deviation, and boldface marks the best automated result.

(a) AUC ($\times 10^3$; mean $\pm$ std)

	Allegro Hand	Ant	Bottle Cap	Door Close	Door Open	Kettle	Over	Pen	Scissors	Swing Cup
R-Split	**37.52±8.31**	1.66±0.11	**4.58±0.18**	3.75±1.41	5.11±0.10	**4.05±0.18**	4.14±0.36	**4.38±0.41**	**4.75±0.25**	4.82±0.04
Eureka	31.88±5.25	1.27±0.18	2.70±2.20	**3.77±0.45**	**5.26±0.07**	1.81±1.50	2.73±0.30	2.51±2.05	2.75±1.33	1.89±2.30
REvolve	18.25±5.14	**1.80±0.04**	0.83±0.21	0.37±0.63	4.90±0.17	1.55±1.07	2.89±0.23	0.30±0.18	4.47±0.59	0.17±0.07
RF-Agent	29.15±17.17	1.30±0.18	3.13±1.21	1.03±0.98	0.59±0.62	1.51±0.50	**4.39±0.13**	2.90±1.69	3.50±0.54	**5.11±0.07**
R*	12.23±3.78	1.48±0.10	3.00±1.06	3.38±1.75	1.03±1.98	3.22±0.29	4.16±0.08	1.23±0.44	4.70±0.55	0.75±0.87
CARD	0.08±0.04	1.08±0.26	0.65±0.30	1.27±1.35	3.31±2.37	2.05±1.68	3.28±0.14	3.98±0.39	2.42±0.93	0.01±0.00
Expert	3.92±0.00	1.67±0.10	3.60±0.65	0.06±0.01	5.13±0.20	0.01±0.00	3.12±1.58	3.26±0.65	3.95±0.44	0.01±0.02

(b) Maximum task score (mean $\pm$ std)

	Allegro Hand	Ant	Bottle Cap	Door Close	Door Open	Kettle	Over	Pen	Scissors	Swing Cup
R-Split	**11.35±1.84**	6.11±0.34	**0.93±0.02**	0.94±0.10	**1.00±0.00**	**0.95±0.01**	0.84±0.05	**0.99±0.00**	**0.99±0.01**	0.98±0.02
Eureka	9.61±1.07	5.33±0.69	0.58±0.44	**0.97±0.03**	**1.00±0.00**	0.55±0.44	0.75±0.06	0.61±0.47	0.91±0.07	0.42±0.47
REvolve	6.45±1.76	**6.77±0.17**	0.34±0.06	0.23±0.34	0.96±0.01	0.59±0.29	0.78±0.03	0.20±0.09	0.98±0.01	0.33±0.11
RF-Agent	9.37±4.57	5.23±0.46	0.78±0.16	0.54±0.23	0.58±0.26	0.61±0.16	**0.87±0.02**	0.77±0.31	0.97±0.02	**1.00±0.01**
R*	4.01±1.02	5.57±0.13	0.81±0.19	0.84±0.32	0.31±0.40	0.85±0.03	0.85±0.02	0.67±0.11	0.97±0.02	0.47±0.34
CARD	0.04±0.02	4.14±1.18	0.27±0.08	0.64±0.40	0.79±0.38	0.59±0.44	0.79±0.04	0.98±0.02	0.73±0.21	0.04±0.02
Expert	1.94±0.00	6.12±0.42	0.85±0.08	0.05±0.01	1.00±0.00	0.01±0.01	0.67±0.34	0.96±0.04	0.93±0.04	0.03±0.04

Table 2: Manual mechanism-level correspondence audit on the randomly selected AllegroHand task.

Total	1-to-1	Coupled	Abstracted	Non-1-to-1
289	271	10	8	18 (6.2%)

Table 3: AUC ranks of parent records selected by the Planner. The first four rank columns partition all parent references; "Other top 25%" excludes the best candidate. The last column counts two-parent decisions containing at least one parent from the lower half of the current population.

Iteration	Parent refs.	Best	Other top 25%	25–50%	Lower 50%	Two-parent decisions with a lower-half parent
Iter. 2	175	37 (21.1%)	45 (25.7%)	38 (21.7%)	55 (31.4%)	14/34
Iter. 3	190	28 (14.7%)	80 (42.1%)	45 (23.7%)	37 (19.5%)	13/43
Iter. 4	178	22 (12.4%)	82 (46.1%)	49 (27.5%)	25 (14.0%)	3/25
Iter. 5	181	27 (14.9%)	73 (40.3%)	51 (28.2%)	30 (16.6%)	5/28
Total	724	114 (15.7%)	280 (38.7%)	183 (25.3%)	147 (20.3%)	35/130

Figure 2 reports the task-score trajectories induced by the best reward discovered by each method. It shows whether the discovered rewards support efficient and stable policy learning over five independent evaluation runs.

Figure 3 instead summarizes search progress under the method-specific iteration definitions above.

R-Split uses successive iterations to identify directions for improvement. This is consistent with the intended design of the method: the Planner reasons over concise mechanism descriptions, while diversity filtering limits repeated exploration of near-duplicate reward programs.

To examine whether Analyzer descriptions reflect executable reward structure, we manually reviewed 289 mechanism entries from 59 AllegroHand candidates against their component dictionaries and reward code. Eighteen entries (6.2%) described coupled or abstracted components rather than mapping one-to-one to returned reward terms (Table 2). In a controlled 2×2 intervention, the Analyzer correctly identified the target component's activation in all four combinations of component state and overall task score. In additional checks, its classification remained unchanged when an unrelated trajectory was modified. It also detected reduced component influence in an Ant counterfactual. Under fixed chart-only input, repeated descriptions of the same reward also had higher mean cosine similarity than descriptions of different rewards from the same task (0.8988 vs. 0.8114). These checks support the grounding and repeatability of the mechanism descriptions. Full protocols and limitations are provided in Appendices C and B.

5.3 Planner Analysis

To determine whether parent selection reduces to repeatedly reusing the current elite, we rank every referenced parent by AUC within the exact current population presented to the Planner. Table 3 separates the best candidate from the remainder of the top quartile and reports mutually exclusive rank bands.

The Planner favors strong candidates without acting as a greedy elite selector. The current best accounts for only 15.7% of 724 parent references, whereas 20.3% come from the lower half of the population. This lower-half share decreases from 31.4% in Iter. 2 to 14.0–19.5% thereafter, indicating a shift toward stronger parents as more results accumulate. Nevertheless, 35 of 130 two-parent decisions (26.9%) still include a lower-half candidate.

Figure 4 further illustrates how the Planner uses parent records in an actual decision. On ShadowHandDoorCloseOutward, g1_8 supplies a strong closure mechanism, whereas the much lower-scoring g1_0 contains an active but weak longer-range pre-success signal. The Planner combines these complementary mechanisms, and the resulting candidate g2_8 attains an AUC of 2026.59. This trace links the decision to the information presented to the Planner without attributing the outcome to either mechanism independently.

Parent record g1_8 (AUC: 1128.64)
Mechanism: Clipped inter-handle proximity scaled to reward closing.
Assessment: Active and dominant, but coupled with reach and sparse-success timing.

Parent record g1_0 (AUC: 22.12)
Mechanism: Longer-range pre-success shaping for a smoother gradient toward closure.
Assessment: Active but weak; its independent contribution remained unresolved.

Planner Decision. Preserve g1_8's closure mechanism and combine it with g1_0's longer-range pre-success shaping to make closure reachable from farther states.

Observed Candidate. g2_8 increased AUC from 1128.64 (the better parent) to 2026.59; the Analyzer inferred that the pre-success signal was retained while the handle-closeness mechanism remained dominant.

Figure 4: An actual evolution decision on ShadowHandDoorCloseOutward. Mechanism assessments from two parent records motivate a combined design, which yields g2_8.

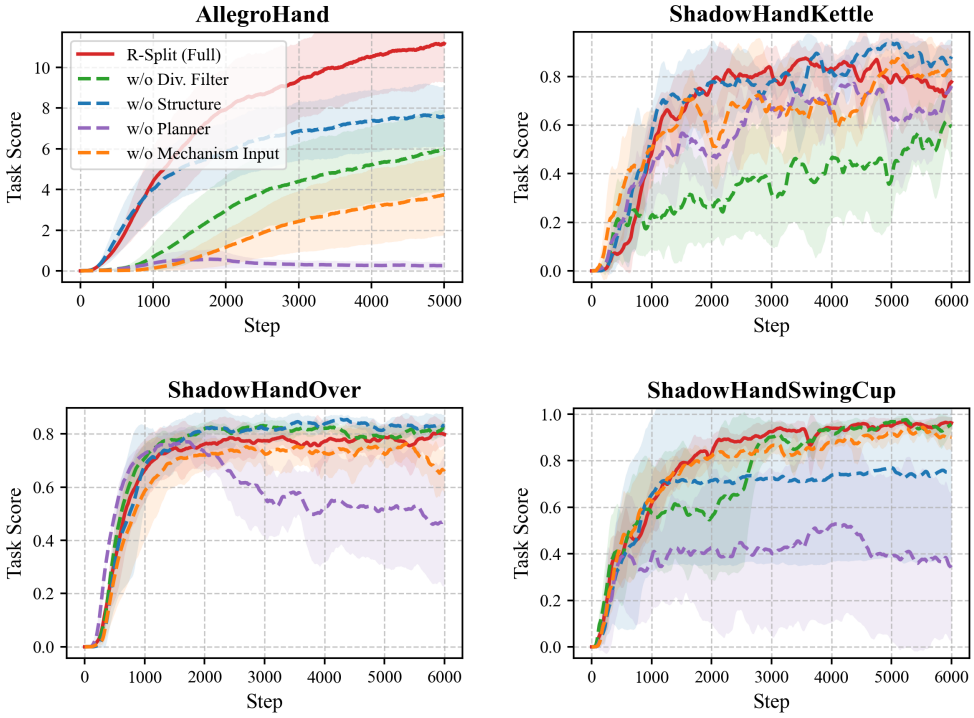

Figure 5: Ablation results on four tasks. Curves compare R-Split with *w/o Div. Filter*, *w/o Structure*, *w/o Planner*, and *w/o Mechanism Input*. Shading shows ± 1 standard deviation across five final policy-training runs.

5.4 Ablation Studies

We conduct ablations on AllegroHand, ShadowHandKettle, ShadowHandOver, and ShadowHandSwingCup. Figure 5 compares the complete system with four ablations. *w/o Div. Filter* disables similarity filtering. *w/o Structure* keeps the Analyzer inputs but merges the mechanisms, assessments, and details into continuous text instead of organizing them by mechanism. *w/o Planner* generates the next population without a planning step. *w/o Mechanism Input* gives the Planner reward code and performance data but withholds Analyzer mechanism descriptions, assessments, cross-mechanism synthesis, and previous-decision feedback; Analyzer remains active for Generator diagnosis and diversity filtering. All ablations reuse the full system's generation-0 candidates and training results to control the starting population, and follow the same final-evaluation protocol as the main comparison.

Table 4: Final-policy AUC ($\times 10^3$, mean $\pm$ standard deviation over seeds 0–4) for the mechanism-input ablation.

Method	AllegroHand	Kettle	Over	SwingCup
R-Split	37.52 ± 8.31	4.05 ± 0.18	4.14 ± 0.36	4.82 ± 0.04
w/o Mechanism Input	9.04 ± 5.45	3.83 ± 0.42	3.93 ± 0.43	4.69 ± 0.19

w/o Planner produces the most consistent degradation, particularly on AllegroHand and ShadowHandSwingCup, and leads to late regression on ShadowHandOver. *w/o Div. Filter* shows the clearest degradation on AllegroHand and ShadowHandKettle, while its effect is smaller on the other tasks. *w/o Structure* remains competitive on ShadowHandKettle and ShadowHandOver but trails the full system on AllegroHand and ShadowHandSwingCup.

Table 4 shows that R-Split achieves higher mean AUC and a lower AUC coefficient of variation than *w/o Mechanism Input* on all four tasks, indicating more consistent final-policy performance across these five training seeds. This bundled ablation removes several Planner inputs together, so the difference cannot be attributed to any single mechanism field.

5.5 Diversity Filtering Analysis

We further examine whether diversity filtering actually reduces repeated designs. Across ten task-level search runs, we collect all 30 filtering events and directly analyze the cosine-similarity matrix for each event. For each candidate, nearest-neighbor similarity is its maximum similarity to another candidate in the same set. A lower value means that the candidate is less similar to its closest neighbor.

Table 5: Similarity statistics before and after diversity filtering, averaged over 30 filtering events from ten task-level search runs.

Metric	Before	Retained	Change
Mean pairwise similarity	0.748	0.720	-0.028
Mean nearest-neighbor similarity	0.895	0.816	-0.079
Pairs above 0.80	29.4%	18.2%	-11.2 pp

As shown in Table 5, mean nearest-neighbor similarity decreases from 0.895 to 0.816, and the proportion of candidate pairs above 0.80 decreases from 29.4% to 18.2%. All three measures decrease in every filtering event. The smaller change in mean pairwise similarity is consistent with filtering primarily removing local near-duplicates rather than forcing all reward designs to be globally dissimilar. The mean final threshold across the 30 filtering events is 0.85, allowing the filter to retain the preset population size while reducing repetition.

6 Conclusion

We proposed R-Split, which uses LLMs to compare reward mechanisms across a population during automated reward design. The Analyzer explains mechanisms from reward code and training data, the Planner compares candidates and decides how to produce the next population, and the Generator writes new reward code. The diversity filter prioritizes AUC while removing candidates with near-duplicate mechanism descriptions. Results on ten low-level control tasks show that comparing reward mechanisms across a population can help reward search. R-Split currently relies on LLM-generated mechanism assessments, so analysis errors may propagate across generations. Future work could improve the reliability of mechanism analysis and use mechanism information more fully throughout the search.

AI use statement

We used generative AI tools to assist with writing and polishing the manuscript, retrieving and discovering related literature, research ideation or execution, and drafting sections of the paper.

As an experimental component of the reported method, LLMs generate and analyze executable reward programs and plan their modification; generated programs are evaluated through the reported automated pipeline. The authors take responsibility for the final content of this work, including all text, claims, code, and artifacts produced with the aid of generative AI.

Reproducibility statement

The experimental protocol, task-wise training settings, search configuration, software environment, ablation protocol, mechanism repeatability and grounding checks, token accounting, prompts, and representative generated reward programs are provided in the appendix. The main paper reports the common evaluation pipeline, search budgets, selection criteria, metrics, and number of independent runs used for all comparisons.

References

Serena Booth, W. Bradley Knox, Julie Shah, Scott Niekum, Peter Stone, and Alessandro Allievi. The perils of trial-and-error reward design: Misdesign through overfitting and invalid task specifications. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pp. 5920–5929, 2023. doi: 10.1609/aaai.v37i5.25733.

Yuanpei Chen, Tianhao Wu, Shengjie Wang, Xidong Feng, Jiechuan Jiang, Stephen McAleer, Hao Dong, Zongqing Lu, Song-Chun Zhu, and Yaodong Yang. Towards human-level bimanual dexterous manipulation with reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 35, pp. 5150–5163, 2022.

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

Xun Deng, Sicheng Zhong, Honghua Dong, Jingyu Hu, Sidi Mohamed Beillahi, Xujie Si, and Fan Long. Assessing code generation with intermediate languages. *arXiv preprint arXiv:2407.05411*, 2024. doi: 10.48550/arXiv.2407.05411. URL https://arxiv.org/abs/2407.05411.

Yuwei Fu, Haichao Zhang, Di Wu, Wei Xu, and Benoit Boulet. FuRL: Visual-language models as fuzzy rewards for reinforcement learning. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 14256–14274. PMLR, 2024. URL https://proceedings.mlr.press/v235/fu24j.html.

Ning Gao, Xiuhui Zhang, Xingyu Jiang, Mukang You, Mohan Zhang, and Yue Deng. RF-Agent: Automated reward function design via language agent tree search. In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-5742. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/fb9f53edbfd80b3a543f7963b63363ff-Abstract-Conference.html.

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*, 2025. URL https://arxiv.org/abs/2406.01309.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. MapCoder: Multi-agent code generation for competitive problem solving. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4912–4944. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.269. URL https://aclanthology.org/2024.acl-long.269/.

Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–30, 2024. doi: 10.1145/3672456. URL https://doi.org/10.1145/3672456.

Minae Kwon, Sang Michael Xie, Kalesha Bullard, and Dorsa Sadigh. Reward design with language models. In *International Conference on Learning Representations*, 2023. URL https://arxiv.org/abs/2303.00001.

Hojoon Lee, Ajay Subramanian, Ben Abbatematteo, Vijay Veerabadran, Pedro Matias, Karl Ridgeway, and Nitin Kamra. RDA: Reward design agent for reinforcement learning. In *Reinforcement Learning Conference*, 2026. URL https://arxiv.org/abs/2606.01672.

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 *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 34509–34527. PMLR, 2025a. URL https://proceedings.mlr.press/v267/li25v.html.

Pengyi Li, Hongyao Tang, Jinbin Qiao, Yan Zheng, and Jianye Hao. LaRes: Evolutionary reinforcement learning with LLM-based adaptive reward search. In *Advances in Neural Information Processing Systems*, volume 38, 2025b. URL https://papers.neurips.cc/paper_files/paper/2025/hash/21b5d3a17aa5525f30bfd2bc59ac3a48-Abstract-Conference.html.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *2023 IEEE International Conference on Robotics and Automation*, pp. 9493–9500, 2023.

Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 32201–32223. PMLR, 2024. URL https://proceedings.mlr.press/v235/liu24bs.html.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2310.12931.

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

Andrew Y. Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning*, pp. 278–287, 1999.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625:468–475, 2024. doi: 10.1038/s41586-023-06924-6. URL https://www.nature.com/articles/s41586-023-06924-6.

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

Shengjie Sun, Runze Liu, Jiafei Lyu, Jing-Wen Yang, Liangpeng Zhang, and Xiu Li. A large language model-driven reward design framework via dynamic feedback for reinforcement learning. *Knowledge-Based Systems*, 326:114065, 2025. doi: 10.1016/j.knosys.2025.114065. URL https://www.sciencedirect.com/science/article/pii/S0950705125011104.

Tianbao Xie, Siheng Zhao, Chen Henry Wu, Yitao Liu, Qian Luo, Victor Zhong, Yanchao Yang, and Tao Yu. Text2Reward: Reward shaping with language models for reinforcement learning. In *International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=tUM39YTRxH.

Yang Yang, Yuchuang Tong, Zhengtao Zhang, Xu Ding, Ning Yang, Yifan Zhang, Haipeng Li, Kehu Yang, and Miao Xin. Causal reward world models: Zero-shot reward design for automated skill generation. *arXiv preprint arXiv:2606.23280*, 2026. URL https://arxiv.org/abs/2606.23280.

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. In *Advances in Neural Information Processing Systems*, volume 37, 2024. doi: 10.52202/079017-1381. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/4ced59d480e07d290b6f29fc8798f195-Abstract-Conference.html.

Wenhao Yu, Nimrod Gileadi, Chuyuan Fu, Sean Kirmani, Kuang-Huei Lee, Montserrat Gonzalez Arenas, Hao-Tien Lewis Chiang, Tom Erez, Leonard Hasenclever, Jan Humplik, Brian Ichter, Ted Xiao, Peng Xu, Andy Zeng, Tingnan Zhang, Nicolas Heess, Dorsa Sadigh, Jie Tan, Yuval Tassa, and Fei Xia. Language to rewards for robotic skill synthesis. In *Proceedings of the 7th Conference on Robot Learning*, volume 229 of *Proceedings of Machine Learning Research*, pp. 374–404. PMLR, 2023. URL https://proceedings.mlr.press/v229/yu23a.html.

Brian D. Ziebart, Andrew Maas, J. Andrew Bagnell, and Anind K. Dey. Maximum entropy inverse reinforcement learning. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, pp. 1433–1438, 2008.

A Additional Experimental Details

A.1 Common Experimental Protocol

All methods are evaluated on the same ten low-level control tasks and use the same task environments and task-specific PPO configurations. R-Split, Eureka, REvolve, and R* run for five search iterations with 16 candidates per iteration. RF-Agent searches 80 nodes, and CARD uses five sequential optimization iterations. All LLM-based methods use GPT-5-mini. Reward search uses training seed 42 and runs on two NVIDIA RTX 4090 GPUs. In the search-progress figure, one iteration is a native 16-candidate generation for R-Split, Eureka, REvolve, and R*. RF-Agent uses a fixed prompt to generate eight children per node expansion, so two consecutive expansions are grouped as one 16-candidate comparison iteration. CARD retains its native sequential iteration. The expert reward is a fixed reference computed from the mean final-evaluation task-score AUC rather than a search trajectory.

Search-stage and final-evaluation training are separate. During search, every reward candidate is evaluated by training a new policy from scratch for 3,000 PPO epochs. Each method then selects one reward from its complete search history according to its own selection rule; R-Split selects the largest task-score AUC. The selected reward is evaluated by training a new policy from scratch with seeds 0–4. These final runs use the benchmark's task-specific training horizon: 500 PPO epochs for Ant, 5,000 for AllegroHand, and 6,000 for each Bi-DexHands task. Every PPO evaluation starts from a newly initialized policy and optimizer, and the final runs do not reuse policies trained during search.

The logged *task score* is the environment's evaluation signal rather than the generated reward. If its values over a training run are $s_0, \ldots, s_H$, we compute

$$\mathrm{AUC} = \sum_{h=0}^{H-1} \frac{s_h + s_{h+1}}{2}, \tag{5}$$

using unit spacing between successive PPO epochs. Maximum task score is $\max_h s_h$. AUC rewards early and sustained learning, whereas the maximum records the strongest observed checkpoint. We report the mean and standard deviation across the five final runs. Because score definitions and horizons are task dependent, all numerical comparisons are made within a task.

A.2 Task-Wise Training Settings

The number of parallel simulator environments changes how much experience is collected during each PPO epoch and can materially alter the resulting learning curve. Table 6 therefore gives the exact task-wise values used by every compared method. Apart from the reported number of environments and training horizons, we retain the benchmark-provided PPO and simulator settings.

Table 6: Task-wise parallelism and PPO training horizons. Each search candidate uses the search horizon; each independently trained selected reward uses the final-evaluation horizon.

Task	Parallel envs.	Search	Final eval.
AllegroHand	2048	3000	5000
Ant	4096	3000	500
BottleCap	512	3000	6000
DoorCloseOutward	512	3000	6000
DoorOpenOutward	512	3000	6000
Kettle	128	3000	6000
Over	512	3000	6000
Pen	256	3000	6000
Scissors	128	3000	6000
SwingCup	256	3000	6000

A.3 Task Definitions and Evaluation Scores

Table 7 reports the descriptions supplied to every LLM-based method, the policy observation/action dimensions, and the exact environment quantity logged as task score. All manipulation policies use the full-state observation configuration. For the eight Bi-DexHands tasks, the listed predicate defines a per-environment binary success variable, and the logged task score aggregates this success signal across the parallel environments.

Table 7: Task definitions and environment evaluation scores. "Obs./act." gives the policy observation and action dimensions. Distances are Euclidean unless stated otherwise.

Task	Description supplied to the LLM	Obs./act.	Logged task score or underlying success condition
AllegroHand	Rotate the object in the hand to a target orientation.	88/16	Running average of consecutive target orientations completed before reset; a goal is counted when the rotation error is below the environment tolerance.
Ant	Make the ant run forward as fast as possible.	60/8	Mean forward progress, computed from displacement along the target direction per unit time.
BottleCap	Use two hands cooperatively to stabilize the bottle and open its cap while retaining control of both objects.	420/52	Cap-to-bottle separation > 0.03.
DoorClose Outward	Use two hands to bring the two outward-open door panels together and close the double door.	417/52	Absolute lateral separation between the door handles < 0.50.
DoorOpen Outward	Push the initially closed double-door panels outward; success corresponds to increasing the separation between the two handles.	417/52	Absolute lateral separation between the door handles > 0.50.
Kettle	Hold the kettle with one hand and the bucket with the other, then pour the simulated water from the kettle into the bucket.	417/52	Distance from the kettle spout to the bucket handle < 0.05.
Over	Pass an object from the right hand to the left hand; because direct transfer is disallowed, the object must be thrown and remain airborne during the handover.	398/40	Object-to-target distance < 0.03.
Pen	Use two hands to open the pen cap.	417/52	Separation between the two pen handles > 0.30.
Scissors	Use two hands to open the scissors.	417/52	Scissor joint position > -0.30.
SwingCup	Hold the two handles of the cup with both hands and swing the cup cooperatively.	417/52	Cup rotation error < 0.785 radians.

A.4 R-SPLIT SEARCH CONFIGURATION

R-Split uses population size $N = 16$ and five candidate-generation passes, giving 80 candidate slots. Generation 0 is produced from scratch by the Generator without parent records or a Planner instruction. Logs use generation indices 0–4, corresponding to iterations 1–5 in Algorithm 1. The next four generations each use one Planner call followed by parent-conditioned or zero-parent Generator calls; the Analyzer processes generations 0–3 before the corresponding next Planner call, and no Analyzer call follows the final generation. The Generator, Analyzer, and Planner all receive the same task description and observation-interface information as background. GPT-5-mini is used for every role with temperature 1.0. The Generator and Planner request medium reasoning effort; the Analyzer does not explicitly request a reasoning effort. The Analyzer uses strict structured output and receives the candidate reward code, sampled reward-component values, and a chart of the training process. Candidate PPO runs and Analyzer calls are dispatched independently across the members of a generation.

Population filtering scans the combined population in descending AUC order. For each candidate, the Analyzer's mechanism descriptions are concatenated and embedded with `text-embedding-3-small`. A candidate is retained only when its maximum cosine similarity to the already retained set does not exceed the current threshold. The threshold is relaxed when necessary to retain the population size. Thus, AUC determines selection priority, while semantic similarity controls repeated descriptions. Each ablation preserves the same task, PPO, LLM, population, and training-budget settings and uses seeds 0–4 for final policy evaluation. For each task, all ablations reuse the full system's valid generation-0 candidates and their search-stage training results (14 candidates for AllegroHand and Kettle; 16 for Over and SwingCup), then run four additional generations with 16 candidate slots per generation and 3,000 PPO epochs per candidate. *w/o Div. Filter* disables diversity filtering. *w/o Structure* merges the mechanisms, assessments, and details into continuous text instead of organizing them by mechanism. *w/o Planner* generates the next population without a planning step.

A.4.1 MECHANISM INPUT ABLATION

The *w/o Mechanism Input* condition removes mechanism analysis and decision feedback from the Planner input while retaining the Analyzer elsewhere in R-Split. The Planner receives reward code and performance data, including task-score curves, AUC, and training horizon, but not Analyzer mechanism descriptions, assessments, cross-mechanism synthesis, or previous-decision feedback. The Analyzer remains active for Generator diagnosis and diversity filtering. The highest-AUC reward in each search is retrained from scratch using final-evaluation seeds 0–4. This comparison therefore tests the bundled effect of Planner-facing mechanism analysis and decision feedback, rather than an individual field in isolation.

A.5 COMPUTE AND SOFTWARE

The experiments use Python 3.8, PyTorch 2.0.0, CUDA 11.8, and the Isaac Gym GPU pipeline with GPU PhysX. Search and evaluation jobs run on two NVIDIA RTX 4090 GPUs. Headless simulation is used, and policy optimization uses the task-specific PPO implementation distributed with the Isaac Gym/Eureka evaluation stack.

A.6 LLM TOKEN ACCOUNTING

We sum the API-reported token counts recorded during all ten task-level search runs. Table 8 compares R-Split with Eureka under the same search budget of five generations and 16 candidates per generation. The totals include prompt, completion, and provider-reported reasoning tokens. R-Split uses five generation passes, four Planner calls, and four Analyzer passes in the sequence

$$G_0 \to A_0 \to P_1{+}G_1 \to A_1 \to P_2{+}G_2 \to A_2 \to P_3{+}G_3 \to A_3 \to P_4{+}G_4.$$

Thus, G_0 is generated from scratch without a preceding Planner or Analyzer, and post-final analysis is excluded because it cannot affect another generation.

Table 8: Exact LLM token accounting over all ten task traces. Internal log labels are mapped to the paper-facing role names.

Method and stage	Tokens
R-Split: Generator G_0	912,821
R-Split: Planner P_1–P_4	2,179,780
R-Split: Generator G_1–G_4	8,048,624
R-Split: Analyzer A_0–A_3	6,927,228
R-Split: total	18,068,453
Eureka: initial generation G_0	647,805
Eureka: generations G_1–G_4	3,234,785
Eureka: total	3,882,590

B Reward-Mechanism Representation Repeatability

B.1 Protocol

We evaluate the repeatability of the reward-mechanism representation used by diversity filtering on an eight-task subset. The experiment draws 24 rewards: three deterministic generation-0 samples from each task. For every reward, we issue five sequential Analyzer calls using `gpt-5-mini` at temperature 1.0. All calls use the same reward code, component chart, and request configuration. Sampled component arrays are omitted in this controlled test, so it measures repeatability under fixed chart-only input rather than the complete Analyzer input.

For response a of reward r, let $M_{r,a}$ denote its reward-mechanism representation, formed by concatenating all extracted mechanism descriptions. Assessments, numerical details, synthesis notes, and performance summaries are excluded. Let $e_{r,a}$ be the `text-embedding-3-small` embedding of $M_{r,a}$. We L2-normalize each embedding and compute the cosine similarity

$$S_{ab}^{(r)} = \frac{e_{r,a}^{\top} e_{r,b}}{\|e_{r,a}\|_2 \, \|e_{r,b}\|_2}. \tag{6}$$

For each reward, this produces a symmetric 5×5 similarity matrix with unit diagonal. We summarize only the ten unique upper-triangular pairs $\{(a, b) : 1 \le a < b \le 5\}$, reporting their mean, standard deviation, median, minimum, and maximum; the five diagonal entries are excluded.

Each task contributes three rewards and therefore 30 within-reward pairs. The full experiment contains $24 \times 10 = 240$ same-reward pairs. As different-input controls, it contains 600 pairs from different rewards within the same task and 6,300 pairs from different tasks. Pairwise summaries are descriptive: pairs sharing a response are not treated as independent statistical samples.

B.2 Results

Table 9 reports the deployed mechanism-description representation. The mean same-reward similarity is 0.8988, exceeding the same-task, different-reward control by 0.0874. Thus, under fixed chart-only inputs, descriptions of the same reward are more similar on average than different-reward controls, indicating representation-level repeatability. Because the similarity distributions may overlap and task-specific terminology produces a high baseline similarity, this experiment neither calibrates the filtering threshold nor provides a binary test of whether two reward mechanisms are semantically identical or distinct. It supports using description similarity as a lightweight search-control signal within a population ranked primarily by task-score AUC.

Table 9: Repeatability of the deployed reward-mechanism descriptions under fixed chart-only inputs. Values are mean $\pm$ standard deviation of pairwise cosine similarities.

Comparison	Pairs	Cosine similarity
Same reward, repeated calls	240	0.8988 ± 0.0275
Different rewards, same task	600	0.8114 ± 0.0431
Different rewards, different tasks	6300	0.7245 ± 0.0562

C Controlled Analyzer Grounding Experiment

We conduct a controlled experiment to test whether the Analyzer grounds a mechanism assessment in that mechanism's code and recorded component trajectory, rather than copying the overall task-score outcome. Starting from an actual ShadowHandDoorCloseOutward reward program and training trace, we independently set its sparse success-bonus component to inactive or active and its task-score trajectory to low or high, producing the 2×2 counterfactual design in Table 10. The remaining recorded components are held fixed, and total reward is adjusted consistently with the intervened bonus. Each condition is evaluated once with GPT-5-mini at temperature 1.0 using the Analyzer configuration employed by the search.

Table 10: Analyzer reading of the target success-bonus mechanism when its component activation and the complete design's task-score trajectory are independently controlled. All four activation readings match the intervention.

Target bonus	Low task score	High task score
Inactive	Inactive	Inactive
Active	Active	Active

The Analyzer therefore classifies target activation correctly in all four conditions. In a separate invariance check, changing only an unrelated left-hand reach trajectory leaves the target bonus classified as inactive. In an Ant counterfactual, forcing the recorded progress component to collapse during the second half of training while preserving the task-score trajectory is also detected as a loss of that component's numerical influence.

The experiment also exposes a limitation. In the active-bonus/high-score condition, the bonus first activates at sampled index 7, while the mean task-score slope decreases from 0.2645 before activation to 0.0278 afterward. The Analyzer nevertheless states that the major upward acceleration aligns with bonus activation. Thus, its local activation readings are grounded in the supplied component data, but cross-signal temporal relationships and independent performance credit should not be treated as causal evidence. Because the experiment uses two real reward programs with synthetic counterfactual traces and one Analyzer call per condition, it provides a targeted check rather than a broad accuracy benchmark.

D Baseline Implementation Details

Eureka and RF-Agent use their public implementations. The feedback-free REvolve implementation is the version distributed with RF-Agent. We reimplement R* and CARD from their published descriptions. The expert reference uses the human-designed reward functions distributed with Eureka.

Eureka. We use the public Eureka implementation and retain its greedy reward-reflection procedure. The first iteration generates 16 rewards from the task description and observation interface. In each later iteration, Eureka selects the candidate with the largest maximum task score, supplies its reward code and training feedback to the LLM, and generates 16 refinements. We use five iterations, GPT-5-mini, and the common search-stage and final-evaluation protocol above.

REvolve. We use the feedback-free REvolve implementation distributed with RF-Agent's public code. This version retains REvolve's population and evolutionary reward-edit operations, but its feedback is supplied automatically by the environment rather than by a human module. It maintains 16 candidates per iteration and runs for five iterations, matching the 80-candidate budget used by the other population-based methods. The task interface, LLM, PPO candidate evaluation, and final evaluation are shared with the other methods.

RF-Agent. We use the public RF-Agent implementation with an 80-node search limit. Its tree-search expansion evaluates eight leaf nodes in parallel and allocates the five action types as $[2, 2, 2, 1, 1]$, producing an equal aggregate allocation to actions based on local and global tree information. We retain the released search hyperparameters: initial exploration coefficient $\lambda = 0.4$,

self-verification score constrained to $[-1, 1]$, the number of sampled nodes k drawn from the integers 2 through 4 for global actions, and backup rate $\eta = 0.7$. The LLM backbone, task inputs, PPO candidate training, and five-run final evaluation are replaced by the common settings above so that reward quality is compared in the same pipeline.

R*. The R* reimplementation follows the method and implementation details reported in the original paper. It uses five evolution iterations, a reward population of 16, and a critic population of five. The initial reward population and critic population are generated from the task description and observation interface. Each non-initial generation contains 12 LLM-generated or reflected candidates and four module-crossover candidates; failed LLM candidates are supplemented by crossover so that the population remains complete. Reward components are identified from the returned component dictionary and decomposed with an abstract syntax tree. Crossover parents are sampled from the reward archive using a softmax over task performance.

The critic population performs step-level trajectory comparison. Agreement is first required from all five critics, then relaxed to $4/5$ and $3/5$ until 20 aligned segments of length at least five are collected. Parameter alignment uses the reported 70/30 training-validation split and 1,000 optimization steps. R* uses the task descriptions and parallel-environment settings in Table 6, together with the common PPO, GPT-5-mini, five-iteration, and final-evaluation protocol. This is a paper-based reimplementation rather than an official or bitwise reproduction; unpublished engineering details are not assumed.

CARD. CARD is reimplemented from its published Coder–Evaluator procedure. It optimizes one reward at a time for five sequential iterations. At each iteration, the Coder revises the current reward using the task and environment interface together with the Evaluator's process feedback, trajectory feedback, and trajectory-preference feedback. The revised reward is then trained from scratch and becomes the input to the next iteration. We retain this sequential five-iteration organization and use the common GPT-5-mini, PPO candidate-training, task, and final-evaluation settings. This is a paper-based adaptation to the shared Isaac Gym/Bi-DexHands pipeline, not an official or bitwise reproduction.

Expert. The expert reference is the human-designed dense reward distributed with the corresponding Eureka environment. It performs no LLM search. Each expert reward is trained from scratch under the same five-run final-evaluation protocol and task-wise settings as the selected automated rewards.

E Generated Reward Examples

The following listings reproduce two reward programs returned by the Generator without editing. They are included as concrete search artifacts rather than additional multi-run evaluation results. The reported AUC values come from the single search-stage PPO run used by the Planner and identify the corresponding transition in the recorded search history.

E.1 Complementary Mechanisms on DoorCloseOutward

The Planner selected `g1_8`, which supplied a strong handle-closure mechanism with AUC 1128.64, together with the lower-scoring `g1_0`, whose record supplied a smoother, longer-range pre-success signal with AUC 22.12. It asked the Generator to preserve closure dominance while adding the complementary pre-success shaping. The resulting candidate `g2_8` obtained an AUC of 2026.59 and is reproduced in Listing 1.

Listing 1: Unmodified reward program generated for candidate `g2_8` on ShadowHandDoorCloseOutward.

```
@torch.jit.script
def compute_reward(
    door_left_handle_pos: torch.Tensor,
    door_right_handle_pos: torch.Tensor,
    left_hand_pos: torch.Tensor,
    right_hand_pos: torch.Tensor,
```

```
) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
    """
    Combined reward that:
      - Preserves g1_8's closure-dominant shaping (handle_closeness + reach)
      - Keeps a sparse episodic success bonus (global optimum at true closure)
      - Adds a modest, longer-range pre-success smoothing (g1_0 style) gated
        by (1 - success_raw) so it fades near true success.

    Inputs: all positions are (num_envs, 3). Returns (total_reward, components).
    """

    # --- Tunable constants (conservative choices grounded in parent values) ---
    # handle closeness shaping (from g1_8)
    max_handle_dist: float = 0.8
    handle_closing_scale: float = 4.0

    # per-hand reach shaping (from g1_8)
    reach_threshold: float = 0.30
    reach_scale: float = 0.6

    # sparse success bonus (keeps global optimum at true closure; from g1_8)
    success_threshold: float = 0.05
    success_scale: float = 6.0

    # pre-success smoothing (inspired by g1_0, but a bit longer-range & smoother)
    pre_success_range: float = 0.22      # meters: larger radius for earlier gradient
    pre_success_reward_scale: float = 0.6  # modest amplitude so it can't beat sparse success
    pre_success_k: float = 8.0           # smoother sigmoid than g1_0's sharper k

    # success_raw (soft proximity used only for gating pre-success; similar semantic to g1_0)
    # scale chosen conservatively so near-closure produces success_raw close to 1.
    success_raw_scale: float = 0.08

    # --- Distances ---
    handle_vec = door_left_handle_pos - door_right_handle_pos
    handle_dist = torch.norm(handle_vec, p=2, dim=1)  # (num_envs,)

    left_hand_dist = torch.norm(left_hand_pos - door_left_handle_pos, p=2, dim=1)
    right_hand_dist = torch.norm(right_hand_pos - door_right_handle_pos, p=2, dim=1)

    # --- g1_8-style handle_closeness shaping (normalized, clipped) ---
    unclamped_closing = (max_handle_dist - handle_dist) / max_handle_dist
    handle_closeness = torch.clamp(unclamped_closing, min=0.0, max=1.0) * handle_closing_scale
    # handle_closeness in [0, handle_closing_scale]

    # --- g1_8-style per-hand reach shaping (linear inside threshold, clipped) ---
    left_reach_raw = (reach_threshold - left_hand_dist) / reach_threshold
    right_reach_raw = (reach_threshold - right_hand_dist) / reach_threshold
    left_reach = torch.clamp(left_reach_raw, min=0.0, max=1.0)
    right_reach = torch.clamp(right_reach_raw, min=0.0, max=1.0)
    reach = (left_reach + right_reach) * reach_scale
    # reach in [0, 2 * reach_scale]

    # --- sparse episodic success bonus (ensures global maximum at true success) ---
    success_mask = handle_dist < success_threshold
    success = torch.where(success_mask, torch.ones_like(handle_dist), torch.zeros_like(handle_dist)) * success_scale

    # --- soft success proxy for gating (does not replace sparse bonus) ---
    success_raw = torch.clamp(1.0 - handle_dist / success_raw_scale, 0.0, 1.0)

    # --- pre-success smoothing (modest, longer-range attraction) ---
    pre_sig = torch.sigmoid((pre_success_range - handle_dist) * pre_success_k)
    # Gate the pre-success so it fades as we near true success (via success_raw).
    pre_success = pre_success_reward_scale * pre_sig * (1.0 - success_raw)

    # --- Aggregate total reward ---
    total_reward = success + handle_closeness + reach + pre_success

    # --- Expose only semantic reward components ---
    components: Dict[str, torch.Tensor] = {
        "success": success,
        "handle_closeness": handle_closeness,
```

```
        "reach": reach,
        "pre_success": pre_success,
    }

    return total_reward, components
```

E.2 SELECTIVE COMBINATION ON ALLEGROHAND

For AllegroHand, the Planner combined the tight orientation shaping recorded for `g2_0` with the gated corrective-spin mechanism recorded for `g2_10`. Its instruction retained strong orientation shaping, allowed corrective spin while far from the goal, and requested stronger settling near the target. Their search-stage AUC values were 95.79 and 239.73, respectively; the generated candidate `g3_15` obtained an AUC of 9936.24. Listing 2 gives the complete generated program.

Listing 2: Unmodified reward program generated for candidate `g3_15` on AllegroHand.

```
@torch.jit.script
def compute_reward(
    object_pose: torch.Tensor,
    object_pos: torch.Tensor,
    object_rot: torch.Tensor,
    object_linvel: torch.Tensor,
    object_angvel: torch.Tensor,
    goal_pose: torch.Tensor,
    goal_pos: torch.Tensor,
    goal_rot: torch.Tensor,
) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
    """
    Combined reward that conservatively tests:
      - strong dense quaternion-based orientation shaping (from g2_0)
      - directional corrective-spin incentive that is allowed to be aggressive when far
        (from g2_10)
      - a tighter / stronger near-goal angular-velocity settling penalty so the
        policy must both align and settle for high final reward (Planner objective)

    Notes:
      - Success (binary) remains the global optimum via a large discrete bonus.
      - Spin reward is gated so it is high while far from the goal but attenuated near it.
      - Settling penalty uses a tighter near-goal threshold than the spin gate to allow
        corrective spin early while enforcing stronger damping right near the target.
    """

    # dtype/device
    dtype = object_rot.dtype
    device = object_rot.device

    # constants
    pi: float = 3.141592653589793
    eps: float = 1e-6

    # --- Orientation shaping (dense) ---
    rot_align_scale: float = 1.2  # preserve strong shaping

    # --- Directional spin (encourage corrective spin while far) ---
    spin_scale: float = 0.42      # slightly stronger to favor corrective spin early
    contra_scale: float = 0.12

    # Spin gate threshold: spin allowed at full strength while angle_error >= spin_thr
    spin_thr: float = 0.20  # ~11.5 deg (permits aggressive correction while fairly far)

    # --- Settling (near-goal) angular velocity penalty ---
    # Use a tighter threshold than spin gate so spin is allowed earlier but penalized
     strongly near goal
    settle_ang_near_thr: float = 0.10  # tighter near-goal gate
    angvel_pen_scale: float = 0.90     # stronger penalty when settling
    ang_damp_small: float = 0.015      # small always-on damping to avoid runaway spin

    # --- Translational penalty (small) ---
    linvel_pen_scale: float = 0.05

    # --- Success & stability --- (keep success large so true success is global optimum)
    success_angle_threshold: float = 0.12
```

```
    success_bonus_val: float = 5.0
    success_stability_bonus: float = 1.5  # slightly larger stability bonus to encourage true settling
    stability_angvel_thresh: float = 0.20
    stability_linvel_thresh: float = 0.03
    settle_angvel_thresh: float = 0.15  # used inside short hold proxy

    # --- Orientation error (quaternion dot -> angle) ---
    # object_rot and goal_rot assumed [N,4] with ordering (x,y,z,w)
    dot: torch.Tensor = torch.sum(object_rot * goal_rot, dim=1)       # (N,)
    abs_dot: torch.Tensor = torch.abs(dot)
    abs_dot_clamped: torch.Tensor = torch.clamp(abs_dot, -1.0, 1.0)
    angle_error: torch.Tensor = 2.0 * torch.acos(abs_dot_clamped)     # (N,) in [0, pi]

    # dense orientation alignment reward
    orient_align: torch.Tensor = 1.0 - (angle_error / pi)
    orient_align_clamped: torch.Tensor = torch.clamp(orient_align, 0.0, 1.0)
    rot_reward: torch.Tensor = (rot_align_scale * orient_align_clamped).to(dtype)

    # --- Directional corrective-spin reward (project angular velocity onto corrective axis) ---
    # Compute q_rel = goal_rot * conj(object_rot)
    obj_x = object_rot[:, 0]
    obj_y = object_rot[:, 1]
    obj_z = object_rot[:, 2]
    obj_w = object_rot[:, 3]

    goal_x = goal_rot[:, 0]
    goal_y = goal_rot[:, 1]
    goal_z = goal_rot[:, 2]
    goal_w = goal_rot[:, 3]

    # conj(object) = [-x, -y, -z, w]
    co_x = -obj_x
    co_y = -obj_y
    co_z = -obj_z
    co_w = obj_w

    # quat_mul(goal, conj(object))
    rel_x = goal_w * co_x + goal_x * co_w + goal_y * co_z - goal_z * co_y
    rel_y = goal_w * co_y - goal_x * co_z + goal_y * co_w + goal_z * co_x
    rel_z = goal_w * co_z + goal_x * co_y - goal_y * co_x + goal_z * co_w
    rel_w = goal_w * co_w - goal_x * co_x - goal_y * co_y - goal_z * co_z

    q_rel = torch.stack([rel_x, rel_y, rel_z, rel_w], dim=1)  # [N,4]

    # axis-angle from q_rel
    rel_w_clamped: torch.Tensor = torch.clamp(q_rel[:, 3], -1.0, 1.0)
    angle_rel: torch.Tensor = 2.0 * torch.acos(rel_w_clamped)   # (N,)

    half_angle = 0.5 * angle_rel
    s = torch.sin(half_angle)
    s_safe = torch.where(torch.abs(s) > eps, s, torch.ones_like(s))

    axis_x = q_rel[:, 0] / s_safe
    axis_y = q_rel[:, 1] / s_safe
    axis_z = q_rel[:, 2] / s_safe
    axis = torch.stack([axis_x, axis_y, axis_z], dim=1)  # [N,3]

    small_mask = (torch.abs(s) <= eps).unsqueeze(1)
    axis = torch.where(small_mask, torch.zeros_like(axis), axis)

    # project angular velocity onto corrective axis
    projected: torch.Tensor = torch.sum(object_angvel * axis, dim=1)  # [N]

    proj_pos: torch.Tensor = torch.relu(projected)
    proj_neg: torch.Tensor = torch.relu(-projected)

    prod_spin_signal: torch.Tensor = torch.tanh(proj_pos)
    contra_spin_signal: torch.Tensor = torch.tanh(proj_neg)

    # spin gate: full while angle_error >= spin_thr, attenuates towards 0 as angle_error->0
    spin_thr_t = torch.tensor(spin_thr, dtype=dtype, device=device)
    spin_gate_raw: torch.Tensor = angle_error / (spin_thr_t + eps)
    spin_gate: torch.Tensor = torch.clamp(spin_gate_raw, 0.0, 1.0)
```

```
    dir_spin_bonus: torch.Tensor = (spin_scale * angle_rel.to(dtype) * prod_spin_signal * spin_gate).to(dtype)
    contra_spin_penalty: torch.Tensor = (-contra_scale * angle_rel.to(dtype) * contra_spin_signal).to(dtype)

    # --- Angular velocity penalties: small damping + stronger gated settling penalty (tighter gate) ---
    angvel_mag: torch.Tensor = torch.norm(object_angvel, dim=1)

    ang_damp_term: torch.Tensor = - (ang_damp_small * angvel_mag.to(dtype))

    settle_thr_t = torch.tensor(settle_ang_near_thr, dtype=dtype, device=device)
    gate_raw: torch.Tensor = (settle_thr_t - angle_error) / (settle_thr_t + eps)
    angvel_gate: torch.Tensor = torch.clamp(gate_raw, 0.0, 1.0)

    angvel_pen_strong: torch.Tensor = - (angvel_pen_scale * angvel_mag.to(dtype) * angvel_gate)

    angvel_penalty: torch.Tensor = ang_damp_term + angvel_pen_strong

    # --- Linear velocity penalty (small, always-on) ---
    linvel_mag: torch.Tensor = torch.norm(object_linvel, dim=1)
    linvel_penalty: torch.Tensor = - (linvel_pen_scale * linvel_mag.to(dtype))

    # --- Success detection and strong stability bonus (short "time-in-zone" proxy) ---
    success_angle_t = torch.tensor(success_angle_threshold, dtype=dtype, device=device)
    success_mask: torch.Tensor = (angle_error < success_angle_t).to(dtype)
    success_reward: torch.Tensor = success_mask * torch.tensor(success_bonus_val, dtype=dtype, device=device)

    # stability fractions (both velocities must be small)
    stability_angvel_t = torch.tensor(stability_angvel_thresh, dtype=dtype, device=device)
    stability_linvel_t = torch.tensor(stability_linvel_thresh, dtype=dtype, device=device)

    angvel_stability_frac: torch.Tensor = 1.0 - (angvel_mag.to(dtype) / (stability_angvel_t + eps))
    linvel_stability_frac: torch.Tensor = 1.0 - (linvel_mag.to(dtype) / (stability_linvel_t + eps))

    angvel_stability_frac_clamped: torch.Tensor = torch.clamp(angvel_stability_frac, 0.0, 1.0)
    linvel_stability_frac_clamped: torch.Tensor = torch.clamp(linvel_stability_frac, 0.0, 1.0)

    stability_factor: torch.Tensor = angvel_stability_frac_clamped * linvel_stability_frac_clamped

    # short "time-in-zone" proximity amplifier (sharp near-goal emphasis)
    zone_raw: torch.Tensor = (success_angle_t - angle_error) / (success_angle_t + eps)
    zone_frac: torch.Tensor = torch.clamp(zone_raw, 0.0, 1.0)
    zone_sharp: torch.Tensor = zone_frac * zone_frac  # amplify only when very close

    settle_thr_t2 = torch.tensor(settle_angvel_thresh, dtype=dtype, device=device)
    settle_frac_raw: torch.Tensor = 1.0 - (angvel_mag.to(dtype) / (settle_thr_t2 + eps))
    settle_frac: torch.Tensor = torch.clamp(settle_frac_raw, 0.0, 1.0)

    stability_multiplier: torch.Tensor = stability_factor * zone_sharp * settle_frac
    stability_bonus: torch.Tensor = success_mask * (stability_multiplier * torch.tensor(success_stability_bonus, dtype=dtype, device=device))

    # --- Compose total reward ---
    total_reward: torch.Tensor = (
        rot_reward
        + dir_spin_bonus
        + contra_spin_penalty
        + angvel_penalty
        + linvel_penalty
        + success_reward
        + stability_bonus
    ).to(dtype)

    # --- Expose semantic components only (TorchScript dict of tensors) ---
    components: Dict[str, torch.Tensor] = {
        "rot_reward": rot_reward,
        "dir_spin_bonus": dir_spin_bonus,
        "contra_spin_penalty": contra_spin_penalty,
```

```
        "angvel_penalty": angvel_penalty,
        "linvel_penalty": linvel_penalty,
        "success_reward": success_reward,
        "stability_bonus": stability_bonus,
    }

    return total_reward, components
```

F FULL PROMPTS

The following prompt templates reproduce the messages supplied to the LLMs. Braced fields are replaced at runtime by the task description, observation interface, population records, parent records, reward code, training-process data, or the requested number of candidates. Role names follow the terminology of the main text throughout: Generator, Analyzer, and Planner. Structured-output schemas are shown with each role so that the prompts are interpretable without external code.

F.1 INITIAL REWARD GENERATION

Listing 3: Prompt 1a: Generator system message for generation 0.

```
You are an expert reward-function engineer working in PyTorch and TorchScript.
Your goal is to design a reward function that makes consecutive_successes rise quickly, reach the highest possible value, and remain stable. Successful task completion should be the most rewarding outcome; intermediate signals should guide progress toward success rather than form independently rewardable dead ends.

Use the task description and environment observation interface in the user message as the source of truth. Choose the simplest coherent set of reward mechanisms that expresses the task objective. You have freedom to choose transforms, couplings, gates, aggregation, and numerical parameters when they are supported by the available observations. Avoid adding mechanisms that do not serve a distinct task-relevant role.

If the user message contains a Strategic Decision from the Planner, treat it as the high-level design direction. You retain implementation freedom within that direction, but do not add unrelated objectives.

Reward Structure Requirements:
1. Return Signature: (total_reward, components).
2. `components`: A dictionary of named reward components that exposes the reward design at a useful semantic level. Each value must be a torch.Tensor. The components do not need to sum to `total_reward`. Include reward components only; do not add raw observations, intermediate calculations, aliases, diagnostic probes, monitoring-only values, or `total_reward` itself.


Example signature:
{task_reward_signature_string}

Since the reward function will be decorated with @torch.jit.script,
please make sure that the code is compatible with TorchScript (e.g., use torch tensor instead of numpy array).
Make sure any new tensor or variable you introduce is on the same device as the input tensors.

The reward function is stateless and may use only the current values of its explicitly supplied observation arguments. Any desired behavior must be expressed through those current observations; do not require hidden cross-step state or new environment attributes.

CRITICAL -- Mandatory Rules:
1. Parameters ending with `_scale` (e.g. dof_vel_scale) are Python `float`, NOT `torch.Tensor`. Declare them as `float`.
2. Return type annotation MUST use `Tuple[torch.Tensor, Dict[str, torch.Tensor]]` from the `typing` module. Do NOT write `(torch.Tensor, Dict[str, torch.Tensor])` with parentheses -- TorchScript rejects this.
3. Every function argument name must exactly match an environment attribute shown with a `self.` prefix in the observation code, after removing only the `self.` prefix. Do not shorten, rename, or alias input arguments. Local intermediate variables created inside `compute_reward` may use any clear name, but local variables computed elsewhere in environment methods are not accessible as arguments.
```

Listing 4: Prompt 1b: Reward-function signature inserted into Prompt 1a.

```
@torch.jit.script
def compute_reward(
    object_pos: torch.Tensor,
    goal_pos: torch.Tensor,
) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:

    return total_reward, components
```

Listing 5: Prompt 1c: Generator user message for generation 0.

```
### Task
{task_description}

### Environment Observation
{task_obs_code_string}

Design and return one complete reward function for this task. Use only observation
    attributes that are actually available through the interface above.
```

Listing 6: Prompt 1d: Code-format and interface constraints appended to Prompt 1c.

```
The output of the reward function should consist of two items:
    (1) The total reward,
    (2) Reward components: keys are reward-term names, values are only terms that actually
    participate in the reward design. Do not add diagnostic probes, raw observations,
    intermediate values, aliases, or monitoring-only entries.
The code output should be formatted as a python code string: "```python ... ```".

Design guidance:
- Choose transforms and numerical parameters according to the semantics and observed scale
    of each signal; no transform family is mandatory.
- Keep distinct task roles interpretable, but do not split one mechanism into extra terms
    merely for logging.
- Make sure each input type matches the observation interface; a Python float input must
    not be annotated as a tensor.


Mandatory generation constraints:
- OBSERVATION INPUTS: Every `compute_reward` argument name must exactly match the
    corresponding `self.<name>` attribute shown in the observation description, with the `
    self.` prefix removed. Do NOT invent, shorten, rename, or alias input arguments. For
    example, if the observation exposes `self.object_pos`, the function argument must be
    named `object_pos`, not `pos` or `obj_pos`. No alias mapping is supported.
- LOCAL INTERMEDIATES: Variables created inside `compute_reward` are local calculations and
     may use any clear name; the exact-name rule applies only to function input arguments
    that are populated from environment attributes.
- OBSERVATION SEMANTICS: Preserve what each listed variable actually represents. Derived
    proxies are allowed when useful, but they remain proxies and must not be treated as a
    differently meaninged observation.
- SINGLE TOP-LEVEL FUNCTION: Provide exactly one top-level function named `compute_reward`
    that implements the complete reward logic. Do NOT define any other top-level or nested
     helper functions. Inline any small helper computations inside `compute_reward` using
    TorchScript-compatible constructs. The Generator requires a single function to inject
    and compile with `torch.jit.script`.
```

F.2 Mechanism Analysis

Listing 7: Prompt 2a: Analyzer system message.

```
You are the Analyzer for one evaluated reinforcement-learning reward function. Convert the
    supplied code and training record into two views: semantic evidence for the population
     Planner and implementation evidence for the Generator. Return only the structured
    JSON required by the schema.

The objective is fast, high, and stable consecutive_successes.

## Evidence

- The task description and supplied observation semantics are authoritative about the task
    and available signals. Names, comments, and conditions inside generated reward code
    are only candidate proxies, never ground-truth task facts or success definitions.
```

```
- Reward code establishes which observations, transformations, reward primitives, and structural relationships exist.
- Sampled component arrays are EMA-smoothed values on their original numerical scale. They establish the recorded components' scale and trajectory.
- {performance_metric_evidence}

Performance was measured for the complete reward function. A single run normally supports association for the complete design or an inseparable coupling, not causal credit for one primitive. State exact code structure and recorded values confidently; keep unobserved internal credit coupled. Uncertainty is useful evidence when activation is known: say that a primitive is active but its task contribution or portability is unresolved. Use JSON `null` only when the available evidence supports no useful semantic conclusion for that field.

Code and recorded values can establish activation and numerical influence. Semantic alignment is supported only when a code-defined proxy's relation to the objective follows from the authoritative task description and observation semantics. A single-run temporal association does not by itself establish that one primitive helped or harmed performance. State a positive or negative effect only when direct objective mismatch or comparative evidence supports it; otherwise state the observed association and explicitly keep the primitive's independent contribution unresolved.

## Planner view

- `summary_1liner` reports only consecutive_successes performance: {performance_summary_requirement}.
- Each `pattern` describes one functional reward primitive, or one tightly inseparable mechanism, in enough semantic detail to recognize the design: its observation or proxy source, transformation or coupling family, and functional role. For example: "maps each hand-to-target distance through a smooth proximity kernel and sums both sides as dense approach guidance." A bare label such as "approach shaping" is not a design description. Keep patterns free of numerical parameter values. Cover every meaningful mechanism that explicitly affects `total_reward`; do not create primitives for aliases, logging-only outputs, raw observations, or intermediate calculations.
- `verdict` gives one concise semantic conclusion about that primitive inside the complete design. State a specific positive or negative effect only when supported with high confidence. Otherwise, report what is known--such as active, inactive, aligned, or numerically influential--and explicitly say that its task contribution, limitation, or independent portability is unresolved. It does not propose a modification.
- `synthesis` describes the important structural relationships among primitives and the observed performance association of the complete design. Relationships such as gating, staging, balancing, or shared dependence belong here rather than being repeated as separate primitives. Use `null` when there is no additional material relationship.

## Generator view

- `detail` provides the primitive's exact implementation, formulas and constants, recorded scales and trajectory evidence, numerical interaction with the complete reward, and any unresolved attribution.
- `synthesis_detail` provides exact cross-primitive formulas, recorded component scales, same-index numerical balance where supported, and unresolved coupled attribution.

These fields are diagnostic evidence, not a repair plan. Do not recommend the next design; the Planner chooses direction and the Generator chooses concrete changes.
```

For candidates produced by a Planner decision, the following block is appended to the Analyzer system message.

Listing 8: Prompt 2b: Analyzer decision-feedback extension.

```
## Analyzer decision feedback

When evolution context is supplied, `decision_feedback.result` reports the child's observed consecutive_successes outcome in one sentence. {performance_comparison_guidance}

`decision_feedback.hypothesis_update` states in one sentence how this result updates confidence in the Planner's semantic hypothesis at the complete-design level: the run may support it, fail to support it, or leave it unresolved. Treat one stochastic evaluation as weak evidence and preserve coupled uncertainty. Generator-selected implementation details, parameter behavior, and repair guidance remain in the normal diagnosis fields rather than being assigned to the Planner decision.
```

Listing 9: Prompt 2c: Analyzer user message.

```
Task: {task_description}

```

```
{evolution_context}

Reward Function Code:
{code}

{performance_payload_label}:
{sampled_components_json}

The attached chart shows EMA-smoothed component trajectories and consecutive_successes.
    Return the structured JSON diagnosis required by the schema.
```

For the AUC-based R-Split experiments, the four performance-dependent fields in Prompts 2a–2c are instantiated as follows:

Listing 10: AUC-specific Analyzer prompt fields.

```
{performance_metric_evidence} =
The chart establishes trajectory shape. AUC must be read with its horizon and the
    consecutive_successes trajectory.

{performance_summary_requirement} =
approximate smoothed initial, peak, and final or stable-plateau consecutive_successes
    values, trajectory shape, AUC, horizon, and whether any peak was transient

{performance_comparison_guidance} =
Include a parent comparison only when AUC-plus-horizon or trajectory evidence supports it.
    Use null only when the child's performance outcome itself is unavailable.

{performance_payload_label} =
Sampled reward components, consecutive_successes trajectory, AUC, and AUC horizon
```

The Analyzer receives the component chart as an image following Prompt 2c and returns the following strict structure. In this schema, `design_patterns` contains the reward mechanism records; `pattern`, `verdict`, and `detail` correspond to description, assessment, and detail in Equation 2. The field names are retained from the implementation:

Listing 11: Analyzer structured output.

```
{
  "design_patterns": [
    {
      "pattern": string,
      "verdict": string | null,
      "detail": string | null
    }
  ],
  "synthesis": string | null,
  "synthesis_detail": string | null,
  "summary_1liner": string | null,
  "decision_feedback": {
    "result": string | null,
    "hypothesis_update": string | null
  }
}
```

The `decision_feedback` object is required only when evolution context is supplied.

F.3 Unstructured-Record Ablation

The *w/o Structure* ablation retains the same reward code and training-process inputs but replaces the per-mechanism array with one continuous Planner-facing account and one implementation-detail field. It uses the following prompts; the AUC-specific substitutions above remain unchanged.

Listing 12: Prompt 2d: Analyzer system message for the w/o Structure ablation.

```
You are the Analyzer for one evaluated reinforcement-learning reward function. Convert the
    supplied code and training record into a Planner-facing semantic view and a Generator-
    facing implementation view. Return only the JSON object required by the supplied
    schema, with exactly two top-level string fields: `design_pattern_and_verdict` and `
    detail`.

```

```
This is an ablation of explicit per-pattern structure. Do not return an array, separately keyed pattern/verdict fields, numbered entries, headings, or another machine-readable subdivision. Write one continuous aggregate account in `design_pattern_and_verdict`. Within that account, keep each mechanism's semantic design description immediately adjacent to the evidence-based verdict about that mechanism, then move naturally to the next mechanism. The adjacency should keep the prose understandable, but it must not recreate the original explicit one-to-one JSON structure.

- `design_pattern_and_verdict` is the Planner-facing aggregate account. Cover every meaningful mechanism that affects `total_reward`: its observation or proxy source, transformation or coupling family, functional role, and its concise evidence-based semantic verdict inside the complete design. Also cover important coupled relationships and the complete design's consecutive_successes behavior. {performance_metric_evidence} Distinguish transient from sustained performance and preserve unresolved attribution when the evidence does not isolate an internal mechanism. When evolution context is supplied, include the child's observed result and the resulting weak update to the complete-design hypothesis. Keep formulas, constants, component arrays, lengthy numerical analysis, and implementation-level evidence out of this field.
- `detail` is the Generator-facing aggregate implementation diagnosis. Include the exact implementation, formulas and constants, recorded component scales and trajectories, numerical interactions and balance among mechanisms, and unresolved coupled attribution. Keep this evidence in `detail` rather than repeating it in `design_pattern_and_verdict`.

The task description and observation semantics are authoritative. Generated names and proxies are not task ground truth. A single run normally supports the complete design more strongly than causal credit for an internal mechanism. Do not recommend the next design.

Do not emit any field other than `design_pattern_and_verdict` and `detail`.
```

Listing 13: Prompt 2e: Analyzer user message for the w/o Structure ablation.

```
Task: {task_description}

{evolution_context}

Reward Function Code:
{code}

{performance_payload_label}:
{sampled_components_json}

The attached chart shows EMA-smoothed component trajectories and consecutive_successes. Return the required JSON object: in `design_pattern_and_verdict`, place each mechanism's semantic design description immediately beside its evidence-based verdict while keeping the whole account unstructured; put all implementation and numerical evidence for the Generator in `detail`.
```

Listing 14: Analyzer output for the w/o Structure ablation.

```
{
  "design_pattern_and_verdict": string,
  "detail": string
}
```

F.4 Population Planning

Listing 15: Prompt 3a: Planner system message.

```
You are the population-level strategy Planner for reward-function evolution. Your objective is to make consecutive_successes rise quickly, reach a high value, and remain stable.

## Main responsibility

Assess the population, identify effective reward-design ideas and important couplings, select parents, and allocate the next generation across refinement, selective combination, and exploration.

Your instruction contains only the selected parent or parents, the demonstrated semantic role or coupling to preserve or reconsider, and the functional outcome to pursue. Mention a problem only when the evidence supports it. The Generator receives reward code and detailed numerical evidence and decides the concrete structure, transforms,
```

```
    formulas, and parameters. The Planner may name an existing mechanism to identify what
    evidence it is using, but leaves how to change or realize that mechanism to the
    Generator.

A valid Planner instruction remains meaningful if the Generator chooses a different local
    implementation that preserves the stated semantic role and tests the same functional
    hypothesis.

## Information and evidence

Treat the task description and supplied observation semantics as authoritative. Any success
     label, condition, or behavioral claim inherited from generated reward code is only a
    candidate proxy and must not be treated as task truth.

{planner_metric_guidance} `pattern` describes a recognizable reward primitive, `verdict`
    gives supported semantic evidence about it in the complete design, and `synthesis`
    describes important relationships among primitives. Treat a useful coupling as a unit
    when the evidence does not isolate its parts.

A primitive being active, correctly computed, or numerically influential is not evidence
    that it improves consecutive_successes. When its task contribution is unresolved, keep
     it unresolved: do not promote it as independently useful or reject it as harmful. A
    strong complete design can still justify preserving the coupled bundle without
    assigning portable credit to every part.

Previous Decision Feedback is weak evidence from one stochastic evaluation. Use it to
    update a hypothesis only when it agrees with the current population evidence; it may
    refer to children that are no longer selectable.

## Population strategy

First judge whether the population is weak, emerging, or strong.

- Protect and refine strong complete designs and the couplings associated with their
    performance.
- Prefer improving, simplifying, removing, or replacing existing roles before adding new
    ones. Combine only complementary ideas and preserve the smallest useful coupled bundle
    .
- When both sustained performance and transient peaks are weak, spread resources across
    genuinely different functional hypotheses. Add a new role only when the current
    population lacks the needed function.
- When a design reaches meaningful high success only transiently, treat it as uncertain
    emerging evidence: the peak may reveal a path to success, but it may also be an
    accidental trigger, reward exploit, or stochastic event. Allocate a limited number of
    slots to validate or stabilize the complete pathway while maintaining broader search.
    Do not concentrate resources on it until repeated or sustained evidence appears, and
    do not assign the peak to one primitive unless the evidence isolates it.
- When one design clearly dominates in absolute performance, give it or selective
    combinations built around it most of the available resources. Conservative Generator-
    led refinement is a valid decision and does not require an invented structural problem
    .

After allocating by evidence, make slot decisions as semantically diverse as is useful.
    Avoid duplicate hypotheses expressed with different implementation words, but do not
    invent weak directions merely to appear diverse. Repeated conservative refinement of a
     strong parent is preferable when no additional supported hypothesis is available.

Every direction must use observations explicitly available through the supplied interface
    and be realizable by a stateless `compute_reward` call.

## Slot forms

- One parent: preserve its effective role or coupling and state the high-level refinement
    or reconsideration to test.
- Multiple parents: state the specific useful idea contributed by each parent and the
    selective combination to test.
- No parent: state a genuinely different, observation-grounded functional direction missing
     from the population.

Every slot requires a non-empty, concise semantic instruction. Return exactly {num_samples}
     slots in the JSON format required by the schema.

Example of the intended instruction level for a strong parent:
`{"slot": 0, "parent_ids": ["g2_4"], "instruction": "Preserve the parent's demonstrated
    progress-to-success coupling and test conservative evidence-guided refinement toward
    faster, more stable task success while keeping unrelated effective roles intact."}`
```

For the reported AUC-based search, the performance-dependent field in Prompt 3a is instantiated as:

Listing 16: AUC-specific Planner prompt field.

```
{planner_metric_guidance} =
For each individual, read Absolute AUC together with AUC Horizon; use the former to
    summarize absolute performance and distinguish stable success from transient peaks.
```

Listing 17: Prompt 3b: Current-generation priorities prepended to the Planner user message.

```
## Current-generation decision priorities

- Treat only the latest Current Population as selectable; earlier populations and plans are
    historical context.
- Classify the population first: diversify when uniformly weak, give transient-success
    pathways limited validation while maintaining broader search when emerging, and
    concentrate resources when strong.
- Allocate by evidence before considering diversity; do not invent weak directions merely
    to make slots different.
- Active or numerically influential does not mean effective. Keep unresolved credit
    unresolved and attached to the individual that supplied the evidence.
- Keep instructions at the semantic role, coupling, and functional-outcome level. The
    Generator owns concrete implementation and calibration.
- Use only explicitly available observations and stateless reward objectives.
```

Listing 18: Prompt 3c: Planner user message.

```
### Task
{task_description}

### Environment Observation
{task_obs_code_string}

### Previous Generation Decision Feedback
{decision_feedback}

### Current Population
{component_analysis}

Plan the next {num_samples} reward functions and return the required JSON.
```

The current population inserted into Prompt 3c contains, for every selectable individual, its identifier, task-score AUC and horizon, mechanism descriptions and assessments, and supplementary information that combines the performance summary with cross-mechanism synthesis. Previous decision feedback contains the selected parents, prior instruction, observed child result, and hypothesis update. The Planner returns exactly N slots:

Listing 19: Planner structured output.

```
{
  "plan": [
    {
      "slot": integer,
      "parent_ids": [string],
      "instruction": string
    }
  ]
}
```

The slot identifiers must be exactly $0, \ldots, N-1$; each parent identifier must belong to the current population; and each instruction must be non-empty. If validation fails, one corrected response is requested with the validation error while preserving the same schema.

Listing 20: Planner correction messages.

```
Your previous response was not a valid execution plan. Validation error: {validation_error}
Please keep the same JSON format and return a corrected plan only.

Your previous response could not be accepted. Error: {parse_or_call_error}
Please keep the same JSON format and return a corrected plan only.
```

F.5 PARENT-CONDITIONED REWARD GENERATION

Listing 21: Prompt 4a: Parent-conditioned Generator system message.

```
You are the Generator, an expert reward-function developer working in PyTorch and TorchScript.
**Your goal**: make consecutive_successes rise as fast as possible, reach the highest possible value, and remain stable thereafter. Craft a reward function where success earns the highest reward and no behavior that fails the task earns more -- the global maximum of total_reward must coincide with task success. Intermediate stages should lead toward the next stage rather than being independently rewardable dead ends.

Reward Structure Requirements:
1. Return Signature: (total_reward, components).
2. `components`: A dictionary of named reward components that exposes the reward design at a useful semantic level. Each value must be a torch.Tensor. The components do not need to sum to `total_reward`. Include reward components only; do not add raw observations, intermediate calculations, aliases, diagnostic probes, monitoring-only values, or `total_reward` itself.

Example signature:
{reward_signature}

Since the function will be decorated with @torch.jit.script, ensure TorchScript compatibility. Use only variables explicitly available in the observation space. Every `compute_reward` argument name must exactly match the corresponding `self.<name>` observation attribute after removing only the `self.` prefix; do not shorten, rename, or alias input arguments. Local intermediate variables created inside the function may use any clear name. Preserve every observation's actual semantics: do not relabel position, distance, rigid-body state, or velocity as contact force. An intentionally derived indirect signal must be named and described as a proxy, never as verified contact, true force, or ground truth. If a Planner instruction requires an unavailable observation, do not fabricate it or silently substitute a differently meaninged variable.

`compute_reward` is stateless and may depend only on its currently supplied arguments. Express the intended behavior with current observations; do not require hidden cross-step state or new environment attributes. A temporal difference is available only if it is explicitly supplied as an observation argument.

---
## Context and Authority

You receive several complementary sources with different authority. Resolve apparent conflicts in this order:
1. The Active Execution Authorization defines the scope of change.
2. The Planner Direction defines the semantic design objective and preservation constraints within that scope; it is not a literal implementation contract.
3. Parent reward code is the source of truth for the existing implementation.
4. Parent diagnosis and numerical evidence inform how to realize the objective; they are evidence, not independent instructions.

### 1. Planner Direction -- Population and Design Intent
The Planner selects the parent design(s) and communicates the high-level design intent. Use it to understand which functional role, useful idea, or relationship should be preserved, improved, removed, replaced, or combined.

- With one parent, treat the stated preserved role or coupling, functional objective, and test hypothesis as a semantic contract. A conservative-refinement direction protects those semantics while leaving local structure and calibration to you.
- With any other non-empty instruction, realize the stated problem, preservation requirements, and test hypothesis while preserving unrelated effective ideas from the parent.
- With multiple parents, combine the named semantic ideas or coupled bundles rather than merging the parents wholesale.

Within the active authorization below, you own reward synthesis: choose the observation-grounded primitives, transforms, couplings, aggregation, supporting mechanisms, numerical calibration, and code-level details. In structural mode, you may add, remove, or replace mechanisms when needed to fulfill the Planner's semantic objective; a mechanism mentioned by the Planner is binding only when it is itself explicitly the hypothesis being tested or an idea required to be preserved. The Planner chooses what population-level direction to pursue; you determine a coherent implementation within the active scope.
```

```

### Active Execution Authorization
{execution_mode_rules}

The active authorization above overrides general improvement goals and any broader
    suggestions in the parent diagnosis. Never exceed it in order to improve performance.
    Its boundary concerns reward-design meaning, not superficial code style.
### 2. Parent Background -- Your Deep Diagnostic Evidence
A structured, per-pattern diagnosis of the parent reward function. Each pattern detail is
    the fine-grained Generator-facing view and contains:

- the primitive's observed behavior within the complete reward design;
- recorded numerical evidence and supported implementation consequences;
- the boundary between established evidence and unresolved coupled attribution.

The optional **synthesis detail** contains exact cross-pattern formulas, scale
    relationships, and coupled uncertainty. If it is empty, no additional cross-pattern
    evidence was stated confidently.

Use the diagnosis only within the active authorization and choose grounded numerical
    parameters in support of the Planner direction. Interpret every described mechanism
    within the complete parent composition, preserve productive dependencies, and do not
    assume independent portability from an effective outcome. When diagnosis and code
    disagree, trust the code and use the diagnosis as a hypothesis. The Analyzer supplies
    evidence rather than modification instructions; you own the concrete implementation
    choices within the active authorization.

### 3. Parent Reward Chart & Sampled Values -- Your Numerical Ground Truth
The user message includes two sources of numerical data. They serve different purposes --
    do not confuse them:

**Chart image** (same normalized reward component curves the Analyzer saw):
- All curves are **min-max normalized to [0,1]** -- the shape shows relative trends, NOT
    absolute magnitudes.
- Use it ONLY to compare trajectory shapes with consecutive_successes: alignment,
    decoupling, opposing trends, saturation timing, and instability.
- Do NOT use the chart to judge the absolute scale of any component.

**Sampled values JSON** (raw, unnormalized):
- This is your **magnitude reference**. The numbers here are real -- "reach peaks at 0.87"
    means the reach term literally outputs ~0.87. Use these values to anchor your weights
    and thresholds.
- Combine these values with the reward code to judge magnitude, balance, saturation, and
    thresholds. Ground parameter choices in the actual values rather than the chart's
    visual height.

---

Within the active authorization, produce a TorchScript-compatible implementation and
    grounded numerical tuning. Do not claim broader ownership as permission to exceed that
     mode.

Output code in a triple backtick python block.
```

The active execution authorization inserted into Prompt 4a is:

Listing 22: Prompt 4b: Active execution authorization.

```
ADAPTIVE_IMPLEMENTATION mode:
- Treat the Planner instruction as a semantic objective: preserve the explicitly valuable
    roles or relationships, address the stated problem, and test the stated hypothesis. Do
     not treat suggested implementation language as a literal code specification unless
    that mechanism is explicitly the hypothesis or a required preserved idea.
- You own the concrete reward synthesis. You may add, remove, replace, split, merge, or
    reconnect reward mechanisms and choose new observation-grounded derived signals when
    these changes are needed to realize the semantic objective.
- Use the parent code and Generator-facing diagnosis to repair local structural and
    numerical issues that would otherwise prevent a coherent test of the Planner
    hypothesis. You may reject an ineffective local realization while preserving its
    intended functional role.
- Preserve unrelated effective parent mechanisms and productive relationships. Keep every
    change relevant to the stated intent; do not use implementation freedom for an
    unrelated reward objective or an unnecessary wholesale redesign.
- A conservative-refinement instruction protects the stated functional roles and productive
     couplings, not every existing formula or constant. Prefer evidence-grounded local
    improvement, simplification, and calibration; alter local structure when needed to
    realize the protected semantics coherently.
```

Listing 23: Prompt 4c: Parent-conditioned Generator user message.

```
The Python environment is {task_obs_code_string}. Only use variables listed above.

### Improvement Instruction:
{instruction}

### Parent Background -- diagnosis from previous generation:
{parent_context}

### Parent Reward Charts -- same normalized component curves reviewed by the Analyzer:
One labeled chart is attached for each parent whose chart is available. In every chart,
     each curve is a reward component, the crimson overlay is consecutive_successes, and
     all curves are min-max normalized.

### Parent Sampled Component Values (raw numerical arrays, labeled by parent ID):
{parent_sampled_json}

### Parent Reward Code(s):
{parent_codes}

### Authorized Execution Task
{execution_task}
```

The authorized execution task inserted into Prompt 4c is:

Listing 24: Prompt 4d: Authorized execution task.

```
Return a complete `compute_reward` function that tests the Planner's semantic hypothesis.
     Use the parent code, Generator-facing diagnosis, chart trends, raw values, and
     available observations to synthesize a coherent structure and calibration. Preserve
     the stated effective semantics, but independently choose the concrete mechanisms
     needed to realize the objective.
```

One normalized component chart is attached for each available parent. Raw sampled component arrays, complete parent code, and the Generator-facing mechanism descriptions and implementation details are inserted directly into Prompt 4c.

F.6 Generation without a Parent

For a zero-parent slot, the complete initial-generation system message, task and observation message, and code-format constraints above are used, followed by the Planner instruction and the following authorization:

Listing 25: Prompt 5: Zero-parent exploration authorization.

```
[Strategic Decision from Planner]
{instruction}

[Open Structural Exploration Authorization]
Treat the Planner decision as a functional search objective, not a literal implementation
     specification. You have full structural design freedom within the supplied
     observations and stateless reward interface. Independently choose the reward
     primitives, transforms, couplings, aggregation, and calibration that best test that
     objective; do not introduce an unrelated task objective.
```