MLREF: EFFICIENT MODULE REUSE FOR REWARD DESIGN IN REINFORCEMENT LEARNING VIA LARGE LANGUAGE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Reward function design remains a bottleneck in reinforcement learning. While large language models (LLMs) have enabled automated reward generation, existing methods generate and revise reward functions as monolithic programs, making it difficult to reliably preserve and reuse effective components discovered in earlier iterations, leading to unstable performance across iterations. To address this, we propose **M**odule-**L**evel **R**eward **E**volution **F**ramework (MLREF). At the core of MLREF is a *module pool*, a persistent repository of reusable reward components. MLREF treats the module pool as the primary optimization object: the pool evolves across iterations by accumulating successful modules, refining underperforming ones, and reusing proven components, while reward functions are constructed as linear combinations of modules drawn from this pool. To drive this evolution, MLREF integrates three mechanisms: reflection-based refinement, hybrid credit assignment, and a merge strategy with rollback, which together improve the effectiveness and robustness of reward optimization. Experiments on 17 tasks spanning locomotion and dexterous manipulation show that MLREF outperforms strong baselines by 25.2% on the Isaac Gym suite and 6.6% on the Bi-DexHands suite, with more stable optimization dynamics.

1 INTRODUCTION

Reinforcement learning (RL) has achieved remarkable success in various domains (Goldwaser & Thielscher, 2020; Zhu & Zhang, 2021; Elguea-Aguinaco et al., 2023; Radosavovic et al., 2024). However, its performance critically depends on the reward function, and designing an effective one remains a fundamental challenge. Sparse rewards are easy to define but often provide insufficient learning signals (Stanton & Clune, 2018; Hare, 2019), whereas dense rewards typically require substantial domain expertise and engineering effort (Sutton & Barto, 1998; Eschmann, 2021).

Recent advances in large language models (LLMs) have provided new opportunities for automating reward design, leveraging their strong capabilities in reasoning (Wei et al., 2022; Zhang et al., 2024), instruction following (Ouyang et al., 2022), and code generation (Chen et al., 2021). However, directly generated rewards often suffer from hallucination, syntactic errors, and semantic misalignment with task objectives, limiting their reliability in practical RL scenarios (Ma et al., 2024; Sun et al., 2025a). These challenges motivate the development of iterative refinement frameworks, where LLMs progressively improve reward functions based on feedback from RL training.

To improve the quality of LLM-generated rewards, numerous LLM-based harnesses and workflows have been developed to iteratively refine reward functions through interaction with RL training feedback. However, existing approaches predominantly optimize rewards at the function level by generating or modifying monolithic reward programs, e.g., EUREKA (Ma et al., 2024), R^* (Li et al., 2025b), and RF-Agent (Gao et al., 2026). Without module-level tracking, credit assignment, and semantic recombination, these methods struggle to reuse effective reward components, resulting in redundant and inefficient search.

Although promising, module-level reward optimization introduces several unique challenges beyond function-level optimization. In contrast to optimizing a complete reward program, module-level refinement requires identifying the semantic roles of individual reward components, determining their

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.

contribution to policy improvement, and effectively coordinating interactions among different modules. Moreover, maintaining a reward pool requires a tradeoff between preserving previously discovered effective modules and exploring new reward components, as naive accumulation may introduce redundancy or conflicting incentives. Therefore, designing a module-level reward optimization framework is a more challenging meta-level optimization problem: beyond leveraging holistic RL feedback for reward evaluation, it must attribute performance changes to individual modules and efficiently discover, refine, and reuse effective reward components.

To address these limitations, we propose **M**odule-**L**evel **R**eward **E**volution **F**ramework (MLREF). At the core of MLREF is a *module pool*, a persistent repository consisting of multiple reusable reward components, each targeting a specific aspect of the task. This pool evolves across iterations by accumulating successful modules, refining underperforming ones, and reusing proven components for future construction. At each iteration, reward functions are assembled as linear combinations of modules drawn from the pool. The framework integrates three mechanisms: (i) *reflection-based refinement* that leverages task analysis and historical feedback to guide module design; (ii) *hybrid credit assignment* that derives per-module weights from empirical RL training signals and the LLM's semantic judgment; and (iii) a *merge strategy with rollback* that consolidates successful modules across parallel samples while recovering from failures. Together, these mechanisms enable stable, empirically grounded pool evolution: each module's design is informed by task analysis and historical feedback, its contribution is credited empirically rather than assumed, and successes are consolidated while failures are rolled back, so rewards improve steadily rather than oscillate across iterations.

We validate MLREF on 17 tasks from Isaac Gym and Bi-DexHands, spanning locomotion and dexterous manipulation. Experiments show that MLREF outperforms state-of-the-art LLM-based reward design methods by 25.2% on the Isaac Gym suite and 6.6% on the Bi-DexHands suite, and ablation studies verify the contributions of individual components.

Our contributions are threefold:

- **Diagnosing a structural bottleneck.** Function-level reward optimization gives reward components no identity, so each iteration must rediscover them from scratch, leading to unstable performance and inefficient search.
- **Reframing reward design as module pool evolution.** We introduce the *module pool*, a persistent repository tracking reusable modules, and make it the optimization object: reward functions become disposable linear combinations, while effective modules accumulate and are reused across iterations. We further develop three complementary mechanisms that together enable stable, directed pool evolution.
- **State-of-the-art results with markedly more stable optimization.** Across all 17 tasks, MLREF achieves the best average performance on both suites and, more importantly, sustains steadily improving trajectories where prior methods suffer severe oscillation, as confirmed by both ablation and evolution analysis.

2 Related Work

Before the advent of LLMs, reward design primarily relied on inverse RL (Adams et al., 2022) and preference-based RL (Christiano et al., 2017); both require substantial human effort and generalize poorly across tasks. Early LLM-based work used LLMs as black-box generators queried per timestep (Kwon et al., 2023; Du et al., 2023), which is computationally prohibitive, or generated code-form rewards without iterative refinement (Yu et al., 2023). Later methods introduced iterative refinement guided by human feedback on visualized behaviors (Xie et al., 2024; Guo et al., 2024; Hazra et al., 2025), but still depend on manual intervention, limiting scalability. EUREKA (Ma et al., 2024) established the first fully automated pipeline for iterative reward optimization. Building on this, R^* (Li et al., 2025b) introduced AST-based crossover with preference-based parameter tuning; CARD (Sun et al., 2025a) applied heuristic pre-screening before costly RL training; RF-Agent (Gao et al., 2026) adapted MCTS for reward space search; and FORGE (Fan & Du, 2025) incorporated chain-of-thought reasoning and systematic crossover. LLM-based reward design has also been explored in game-playing (Wu et al., 2023; Li et al., 2024; Yifan et al., 2025), multi-agent RL (Li et al., 2025a;c; Wei et al., 2026), and offline imitation learning (Sun et al., 2025b). None

of these methods, however, maintain an explicit module pool for systematic reuse or account for LLM-induced variance in evaluation.

In contrast, MLREF is the first to maintain a persistent module pool for systematic module-level accumulation, refinement, and reuse throughout reward evolution.

3 PRELIMINARIES

3.1 MARKOV DECISION PROCESS AND REINFORCEMENT LEARNING

We consider sequential decision-making problems formalized as Markov Decision Processes (MDPs) (Puterman, 1990), defined by $(\mathcal{S}, \mathcal{A}, P, R, \gamma, \rho_0)$ with state space $\mathcal{S}$, action space $\mathcal{A}$, transition kernel P, reward function $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, discount factor γ, and initial state distribution ρ_0. A reinforcement learning algorithm seeks a policy $\pi : \mathcal{S} \to \Delta(\mathcal{A})$ that maximizes the expected discounted return $J(\pi; R) = \mathbb{E}_{\pi,P,\rho_0}\big[\sum_t \gamma^t R(s_t, a_t)\big]$. Because R is the sole task-specific signal in this objective, the choice of reward function entirely determines the behavior that is learned.

3.2 REWARD DESIGN PROBLEM

Based on the MDP defined above, a reward design problem (RDP) (Lewis et al., 2010) is characterized by the tuple $\langle \mathcal{E}, \mathcal{R}, \mathscr{A}, F \rangle$, where $\mathcal{E} = (\mathcal{S}, \mathcal{A}, P, \gamma, \rho_0)$ denotes the environment, $\mathcal{R}$ is the space of candidate reward functions, $\mathscr{A}$ is the policy optimization algorithm, and $F : \Pi \to \mathbb{R}$ is the fitness function that evaluates the task performance of a policy, with Π denoting the space of policies. For each reward function $R \in \mathcal{R}$, let $\pi_R = \mathscr{A}(\mathcal{E}, R) \in \Pi$ denote the policy obtained by optimizing R in $\mathcal{E}$. The goal of an RDP is to identify a reward function whose resulting policy maximizes the fitness: $R^* = \arg\max_{R \in \mathcal{R}} F(\pi_R)$. We refer to $F(\pi_R)$ as the performance of the reward function R throughout the paper.

4 METHOD

4.1 OVERVIEW

Inspired by prior work (Ma et al., 2024), MLREF models a reward function as a linear combination of reward modules, where each module targets a specific aspect of the task and, when combined, forms a robust reward function that balances multiple desiderata. Formally, a reward function R is constructed from a set of K modules $\{m_1, m_2, \ldots, m_K\}$ with corresponding weights $\{w_1, w_2, \ldots, w_K\}$:

$$R(s, a) = \sum_{k=1}^{K} w_k \cdot m_k(s, a). \tag{1}$$

To fully leverage the potential of these modules, MLREF introduces a *module pool*, a persistent repository where modules are accumulated, refined, and recombined across iterations. Figure 1 illustrates the full pipeline and Algorithm 1 presents the corresponding procedure; the pipeline consists of two phases: initialization and iterative optimization. In the initialization phase, the LLM performs initial reflection on the task description and environment code (steps 1–2), then generates N diverse module pool variants (step 3). In the iterative optimization phase, each variant constructs a reward function via weighted linear combination (step 4), which is evaluated through RL training (step 5) to obtain its performance F_i. A pool merge step consolidates modules from successful variants and rolls back upon failure based on the training feedback (steps 6–7). The LLM then conducts feedback reflection to analyze training outcomes and generates refined pool variants for the next iteration (steps 8–9). Complete LLM prompt templates are provided in the Appendix.

The key departure from prior work lies in what is optimized. In MLREF, a reward function is only a temporary instance assembled from the pool for RL evaluation; the pool itself is the persistent object that accumulates evidence and improves over iterations. As a result, effective modules persist beyond the function that introduced them and are actively reused in subsequent iterations; training

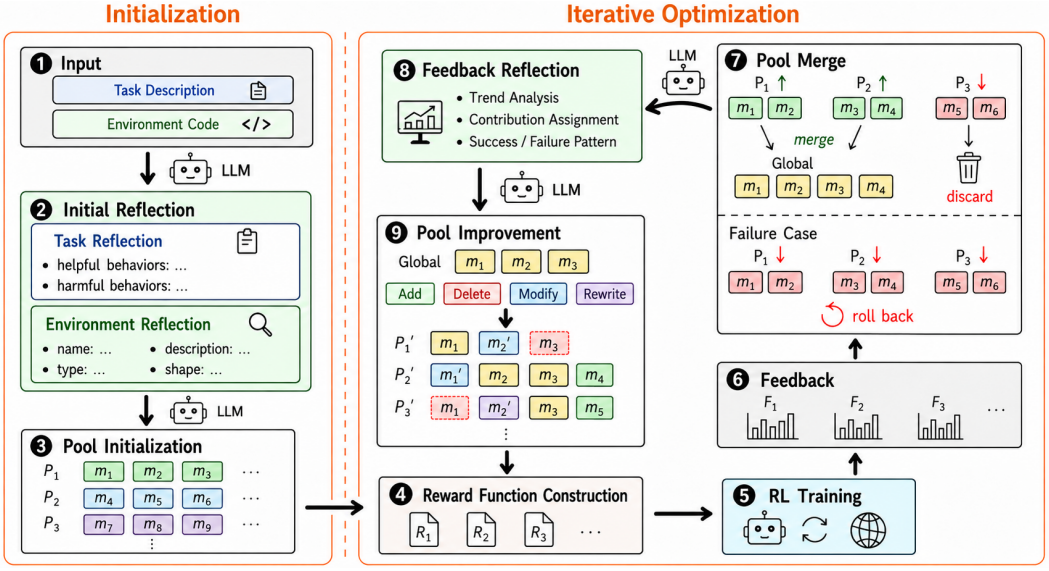

Figure 1: Overview of the MLREF pipeline. In the initialization phase (steps 1–3), the LLM performs task and environment reflection and generates diverse module pool variants. In the iterative optimization phase (steps 4–9), each variant assembles a reward function via weighted linear combination for RL evaluation; a merge strategy with rollback consolidates successful modules; and feedback reflection guides the next iteration of pool improvement.

feedback is applied at the module level, enabling finer-grained and more targeted optimization than function-level rewriting.

4.2 Module Pool

Below we describe the three stages of pool evolution: initialization, improvement, and merge.

4.2.1 Pool Initialization

In the first iteration, MLREF constructs a module pool from scratch. The initialization proceeds in two steps: specification generation and module implementation. A module's specification is its metadata, consisting of the module name, input variable names and types, and a natural-language description of its intended function. In the specification generation step, the LLM generates a set of module specifications that collectively cover the diverse requirements of the task, guided by the task description, environment code, and initial reflection. In the implementation step, the LLM generates Python code for each module based on its specification. Separating specification from implementation improves generation quality by letting the LLM focus on one subtask per response.

4.2.2 Pool Improvement

From the second iteration onward, MLREF refines the existing module pool. Improvement also follows a two-step structure: improvement plan design and improvement plan execution. In the plan design step, the LLM produces an improvement plan based on the training statistics and feedback reflection. We allow four types of operations on modules:

1. **Add.** Introduce a new module by providing its specification (analogous to initialization).
2. **Delete.** Remove a module that is redundant or empirically harmful.
3. **Modify.** Revise a module's code while keeping its specification fixed (e.g., fixing incorrect structure or unsuitable output scale).

Algorithm 1 MLREF Iterative Optimization

Require: Task description T, environment code E, LLM $\mathcal{M}$, RL trainer $\mathcal{A}$, merge strategy MERGE, max iterations I, parallel samples N
Ensure: Best reward function R^*

```
P_global ← ∅, P_prev ← ∅, F_best ← −∞, R* ← ∅
for iter = 1 to I do
  // Stage 1: Module pool generation or improvement
  if iter = 1 then
    ref ← M.REFLECT(T, E) // Initial Reflection
    for i = 1 to N in parallel do
      P_i ← M.INITPOOL(T, E, ref) // Pool Initialization
    end for
  else
    ref ← M.REFLECT(F_prev, F_best, P_prev, P_global) // Feedback Reflection
    for i = 1 to N in parallel do
      P_i ← M.IMPROVEPOOL(T, E, ref, P_global) // Pool Improvement
    end for
  end if
  // Stage 2: Reward construction and RL evaluation
  for i = 1 to N in parallel do
    R_i ← CONSTRUCTREWARD(P_i) // Weighted linear combination (Eq. 1)
    (F_i, T_i) ← A.TRAIN(E, R_i) // Performance and training statistics
    P_i.UPDATECREDIT(M, T_i) // Update hybrid credit
  end for
  // Stage 3: Pool merge and best selection
  P_global ← MERGE(P_global, {P_i}, {F_i}, F_best) // Merge surviving variants or roll back
  i* ← argmax_i F_i // Select the best variant
  F_prev ← F_i*, P_prev ← P_i*
  if F_prev > F_best then
    F_best ← F_prev, R* ← R_i*
  end if
end for
return R*
```

4. **Rewrite.** Replace both the specification and the code of a module. Unlike deletion followed by addition, rewriting explicitly keeps the module's name to preserve continuity, so it can be viewed as a more radical form of modification.

For each operation, the LLM also provides a rationale. In the execution step, the LLM implements the concrete code changes relating to added, modified, and rewritten modules according to the plan. This two-step decomposition again improves output quality by separating planning from coding. By operating at the per-module level, MLREF achieves finer-grained control over reward function optimization than prior function-level approaches.

4.2.3 POOL MERGE

Instead of keeping only the best-performing variant of each iteration (Ma et al., 2024), where the remaining candidates are discarded, MLREF applies a pool merge strategy to consolidate modules from multiple variants. To be specific, variants whose performance F_i falls below the historical best performance F_{best} by more than a relative margin t are discarded. Then all modules from surviving variants are collected into a single set that forms the new module pool; when multiple variants contain the same module, the version from the highest-performing variant is retained. Since all variants are improved from the same base pool and use consistent module names, such a merge is well defined and accumulates the useful changes of several plans at once. On the other hand, if no variant meets the threshold, MLREF rolls back to the previous pool. This rollback mechanism is central to MLREF's iterative stability: it prevents a single poor iteration from destroying the entire optimization trajectory. The failed attempts are recorded and fed into the next iteration of feedback reflection, encouraging the LLM to explore different strategies.

4.3 REFLECTION

Despite decomposing complex processes into multiple steps, LLM outputs can still suffer from hallucination (producing invalid code that crashes RL training) or insufficient reasoning (yielding suboptimal module designs). To improve output stability and quality, we introduce a reflection mechanism inspired by prior work (Li et al., 2025c; Sun et al., 2025b) using chain-of-thought prompting (Wei et al., 2022). As shown in Fig. 1 (steps 2 and 8), MLREF employs two types of reflection: initial reflection before the first iteration, and feedback reflection before each subsequent improvement.

4.3.1 INITIAL REFLECTION

Before the first pool initialization, MLREF conducts initial reflection in two parts. *Task reflection* prompts the LLM to analyze the task description, identifying conditions for success, behaviors likely to help or hinder performance, and non-obvious strategies worth exploring. We explicitly encourage diversity and forbid concrete code generation at this stage. *Environment reflection* extracts relevant observation variables including their names, types, tensor shapes, and descriptions from the environment code, which provides a reference for downstream module design and a validity check against undefined variables. Task and environment reflections are separated to ensure depth in each, and the resulting analysis is reused during all subsequent pool improvement steps.

4.3.2 FEEDBACK REFLECTION

From the second iteration onward, MLREF performs feedback reflection on the training outcome of the previous iteration before planning pool improvements.

If training completed successfully, the LLM receives the historical best pool (with its module composition and training results) alongside the pool of previous iteration, improvement plan, and results. It analyzes reward trends per module, compares module design and composition between the two iterations, and attributes performance changes to specific pool differences. When the previous iteration achieves a new best, the LLM extracts successful design patterns to reinforce; when performance regresses, it diagnoses weaknesses in the improvement plan and proposes new directions.

If training encountered an error, the LLM instead receives the error trace, the previous improvement plan, and the environment code. It locates the error source, analyzes its cause, and suggests concrete measures to avoid similar failures in future iterations.

As with initial reflection, we encourage diverse hypotheses and forbid concrete improvement plans or code at this step, keeping analysis and execution cleanly separated.

4.4 WEIGHT OPTIMIZATION

Assigning appropriate weights to modules is critical: different modules vary in importance and output magnitude, so uniform weighting is suboptimal, while deriving weights from the LLM's judgment alone (Ma et al., 2024; Li et al., 2026) leads to inconsistency across iterations. MLREF addresses this with *hybrid credit assignment* that combines LLM-based semantic judgment with empirical evidence from RL training.

Each module maintains two credits. *LLM credit* captures the LLM's assessment of a module's relevance: after pool initialization or improvement, the LLM is prompted to select modules and propose initial weights, which are normalized to form LLM credits (with unselected modules receiving zero). *Correlation credit* captures the module's empirical contribution: we compute the Pearson correlation between the module's reward sequence and the performance curve during training in the previous iteration, incorporating a time-lag compensation to account for delayed effects of rewards on performance. Here the performance curve is the built-in task-specific per-epoch success metric, and the module reward sequence is averaged over each epoch. Both the raw reward–performance correlation and the correlation of their temporal differences are combined into a single correlation credit. To prevent credit values from oscillating across iterations, both LLM and correlation credits are smoothed via exponential moving average.

To determine the final weights, we normalize the two credits to comparable ranges and fuse them into a composite weight via a weighted sum. For module selection, we apply an upper confidence bound

(UCB) term (Auer et al., 2002) over the composite weight to balance exploitation of high-credit modules with exploration of underused ones. The top-K modules by UCB value are selected, and their composite weights serve directly as w_k in the linear combination (Eq. 1). The full mathematical formulation is provided in the Appendix.

5 EXPERIMENTS

5.1 EXPERIMENTAL SETUP

We evaluate MLREF on 17 representative tasks from Isaac Gym (Makoviychuk et al., 2021) and Bi-DexHands (Chen et al., 2022), covering locomotion and dexterous manipulation challenges (see Table 1 for the full list) to answer the following questions: (i) How does MLREF perform compared to state-of-the-art baselines? (ii) How do individual components of MLREF contribute to its performance? (iii) How stable is the reward evolution process under MLREF?

We compare against two state-of-the-art baselines:

EUREKA (Ma et al., 2024) generates reward function variants at each iteration by prompting the LLM to improve upon the previous best candidate, using the task description, environment code, and training feedback. We follow the original configuration: 5 iterations with 16 samples per iteration.

RF-Agent (Gao et al., 2026) maintains a Monte Carlo tree over the reward space, selecting promising nodes for targeted optimization, evaluating them via RL, and back-propagating the results. We set the number of MCTS updates to 80, matching the total RL training budget of EUREKA.

For MLREF, to mitigate the randomness introduced by the LLM, we run each task for 3 independent complete pipeline executions and return the reward function from the best execution during the iterative process. Each execution consists of $I = 9$ iterations with $N = 3$ parallel module pool variants per iteration, yielding a total RL training budget comparable to the baselines.

For each algorithm on each task, RL training runs for 3,000 epochs within each iteration. After the final reward function is produced, we evaluate it by training a policy from scratch for 20,000 epochs using PPO (Schulman et al., 2017) with 5 independent random seeds, following the RL configuration of EUREKA (Ma et al., 2024).

All LLM calls use the DeepSeek-V4-Flash model (Xu et al., 2026) in reasoning mode via the official API. Full hyperparameter configurations and LLM prompt templates are provided in the Appendix. The environment setting follows the configuration of EUREKA (Ma et al., 2024).

5.2 MAIN RESULTS

To answer Q1, Table 1 compares MLREF with EUREKA and RF-Agent across all 17 tasks. Results are reported as the mean and standard deviation of the task-specific success metrics over five evaluation seeds. For the Isaac Gym suite, we further report the normalized score $\frac{\text{raw}-\text{sparse}}{\text{human}-\text{sparse}}$, where the sparse and human references are taken from RF-Agent (Gao et al., 2026). We group tasks by suite because they use different performance scales and evaluation protocols.

Across the 17 tasks, MLREF achieves the best average performance on both suites. On the Isaac Gym suite, MLREF attains the highest mean on 4 of 7 tasks, with an average normalized score of 3.288, representing a 25.2% improvement over the best baseline. The gain is most striking on Franka Cabinet, where MLREF approaches the maximum possible score (0.997 vs. 0.701 of the nearest baseline). On the Bi-DexHands suite, although MLREF obtains the best individual result on only 2 of 10 tasks, its average performance (0.708) is the highest among all methods, yielding a 6.6% improvement over the best baseline, with particularly strong results on Block Stack and Lift Underarm. These results demonstrate that systematic module-level management via the module pool yields more effective reward functions than function-level optimization across both suites.

5.3 ABLATION STUDY

To answer Q2, we conduct ablation experiments on three representative tasks (Ant, Block Stack, Bottle Cap) by removing individual components from the full MLREF framework:

Task	Domain	EUREKA	RF-Agent	MLREF (Ours)
Isaac Gym Suite (4 locomotion tasks + 3 dexterous manipulation tasks)				
Allegro Hand	Manip.	25.48 ± 1.27	27.41 ± 1.90	**28.89 ± 1.36**
Ant	Loco.	12.35 ± 1.29	12.34 ± 0.70	**12.47 ± 0.30**
ANYmal	Loco.	**-0.0057 ± 0.0021**	-0.0064 ± 0.0010	-0.0073 ± 0.0019
Franka Cabinet	Manip.	0.397 ± 0.202	0.701 ± 0.200	**0.997 ± 0.006**
Humanoid	Loco.	**7.95 ± 0.87**	5.56 ± 1.63	7.40 ± 1.12
Quadcopter	Loco.	-0.042 ± 0.007	-0.277 ± 0.485	**-0.040 ± 0.011**
Shadow Hand	Manip.	**17.11 ± 2.17**	16.53 ± 1.65	11.75 ± 9.69
Avg. Normalized Score		2.135	2.625	**3.288** (+25.2%)
Bi-DexHands Suite (10 dexterous manipulation tasks)				
Block Stack	Manip.	0.116 ± 0.050	0.139 ± 0.096	**0.313 ± 0.244**
Bottle Cap	Manip.	**0.994 ± 0.004**	0.992 ± 0.010	0.987 ± 0.012
Catch Abreast	Manip.	**0.718 ± 0.066**	0.657 ± 0.047	0.644 ± 0.059
Catch Underarm	Manip.	**0.910 ± 0.010**	0.909 ± 0.016	0.871 ± 0.033
Door Close Outward	Manip.	0.266 ± 0.129	**0.415 ± 0.265**	0.256 ± 0.084
Grasp and Place	Manip.	**0.710 ± 0.345**	0.182 ± 0.130	0.475 ± 0.040
Kettle	Manip.	0.441 ± 0.376	**1.000 ± 0.000**	0.901 ± 0.198
Lift Underarm	Manip.	0.569 ± 0.465	0.512 ± 0.394	**0.930 ± 0.022**
Over	Manip.	0.968 ± 0.007	**0.985 ± 0.002**	0.966 ± 0.011
Swing Cup	Manip.	**0.945 ± 0.093**	0.721 ± 0.363	0.736 ± 0.294
Avg. Performance		0.664	0.651	**0.708** (+6.6%)

Table 1: Performance comparison across 17 tasks, grouped by benchmark suite. "Loco." and "Manip." denote locomotion and dexterous manipulation. Each cell reports the mean and standard deviation of the task-specific success metrics over five evaluation seeds. The percentage improvement is relative to the best baseline. Bold indicates the best mean result. Higher values indicate better performance.

Method	Ant	Block Stack	Bottle Cap	Avg. Retention
Full MLREF	**12.47 ± 0.30**	**0.313 ± 0.244**	0.987 ± 0.012	**100.0%**
No Pool	11.51 ± 0.64	0.085 ± 0.040	**0.988 ± 0.010**	73.2%
No Reflection	9.58 ± 0.55	0.057 ± 0.032	0.639 ± 0.401	53.3%
No Credit Assign.	11.68 ± 1.14	0.044 ± 0.025	0.829 ± 0.338	63.9%

Table 2: Ablation results of MLREF components on three representative tasks. Avg. retention is relative to the full MLREF framework. Best results are bold.

1. **No Pool**: remove the module pool; at each iteration, the LLM directly generates a complete reward function, analogous to EUREKA.
2. **No Reflection**: remove both initial and feedback reflection; the LLM generates and improves pools directly from the task description and raw training feedback.
3. **No Credit Assign.**: remove hybrid credit assignment; the LLM selects modules and assigns weights by semantic judgment alone.

Table 2 reports the results. All other experimental settings remain identical to the full MLREF configuration.

The full MLREF framework achieves the best or near-best performance across all tasks. Disabling reflection incurs the largest drop among the ablations (retaining 53.3%), highlighting its critical role in stabilizing optimization. The No Pool variant matches the full framework on Bottle Cap (0.988 vs. 0.987) due to serendipitous exploration, but without the pool, successful designs cannot be stably inherited. Removing hybrid credit assignment also degrades performance, confirming that empirical credit complements the LLM's semantic judgment. We also observed that only the full

MLREF framework consistently generalized from validation to test across all three tasks: the best reward during the iterative evolution remained the best in the final evaluation. In contrast, the ablated variants frequently produced rewards that scored well during validation but underperformed at test, suggesting a tendency to overfit when individual components are removed.

5.4 Iterative Dynamics

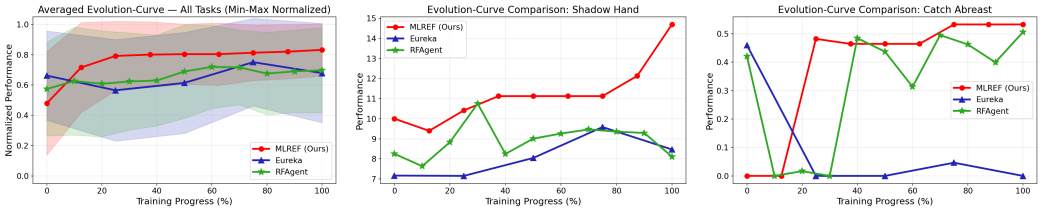

Figure 2: Performance evolution across iterations (best sample per iteration). Left: all-task average. Middle: Shadow Hand. Right: Catch Abreast. Shaded areas denote standard deviation.

To answer Q3, Figure 2 plots the best sample per iteration over optimization progress, averaged across tasks (left) and for two representative tasks (middle, right). For MLREF this is the best pool variant across the 3 runs, for RF-Agent the best expanded MCTS node, and for EUREKA the best of its 16 samples. Since iteration counts differ, we normalize the horizontal axis to $[0, 1]$. For the all-task average, per-task performance is min-max normalized so each task contributes equally.

On the all-task average, EUREKA benefits from large sample counts in early iterations but struggles to make further improvements thereafter, while MLREF, starting from a lower initial point, exhibits consistent improvement through modular directed optimization; RF-Agent's MCTS-based strategy yields intermediate behavior. On Shadow Hand and Catch Abreast, baselines suffer from severe performance oscillation. In particular, EUREKA degrades sharply after a strong first iteration on Catch Abreast. By contrast, MLREF maintains stable trajectories through its rollback mechanism while continuing to improve via iterative refinement, demonstrating strong optimization stability. Complete evolution curves for all 17 tasks are provided in the Appendix.

5.5 Discussion

Several limitations suggest directions for future work. First, on sparse-reward manipulation tasks such as Block Stack, MLREF's absolute performance remains modest, suggesting stronger early exploration and progressive exploitation could improve the balance. Second, all experiments use a single LLM backbone, so generality across models remains to be established. Finally, multi-LLM collaboration to leverage diverse reasoning and knowledge represents a promising direction.

6 Conclusion

We presented MLREF, a module pool-based framework that optimizes reusable reward modules through systematic pool-level operations. Unlike prior function-level approaches, MLREF's module pool enables persistent accumulation, refinement, and reuse of reward components across iterations, supported by reflection, hybrid credit assignment, and a rollback-equipped merge strategy that together achieve iterative stability.

Experiments on 17 tasks spanning locomotion and dexterous manipulation show that MLREF outperforms state-of-the-art LLM-based reward design baselines, improving by 25.2% on the Isaac Gym suite and 6.6% on the Bi-DexHands suite. Ablations confirm each component's contribution, with reflection particularly critical. Evolution analysis further shows that MLREF maintains stable optimization trajectories through its rollback mechanism.

AI USE STATEMENT

In this work, we used generative AI tools for assisting with translation. We have not used generative AI tools for generating synthetic data sets, helping develop theoretical models or conceptual frameworks, designing or providing feedback on research methodology or experiments, implementing methods, cleaning and reformatting datasets, supporting qualitative and thematic data analysis, or interpreting results. In particular, formulating mathematical claims, providing critical ingredients for proving mathematical claims, assisting in the writing of proofs, and proposing or refining hypotheses are not applicable to this work. Additionally, we used generative AI tools for editing software code, drafting parts of a research paper, editing a research paper to improve readability, suggesting a structure for a research paper, and proposing a title for a research paper. We have reviewed all AI-assisted work. LLM-generated code was verified and tested for correctness by the first author. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

This work introduces an LLM-driven reward design framework and reports exclusively empirical results in publicly available simulated environments. It involves no new datasets and no theoretical claims requiring proof. All components needed to reproduce our results are specified in the main paper and the appendix. A complete specification of MLREF, including each of its components, is given in Section 4 together with Algorithm 1. The weight optimization algorithm is fully specified in Appendix A. The experiment settings are listed in Section 5, and the hyperparameters are listed in Table 3 (Appendix B). All prompts used to query the LLM are reproduced verbatim in Appendix D. The per-task and suite-level evolution curves that support our stability claims are reported in Appendix C.

REFERENCES

Stephen Adams, Tyler Cody, and Peter A Beling. A survey of inverse reinforcement learning. *Artificial Intelligence Review*, 55(6):4307–4346, 2022.

Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2):235–256, 2002.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Yuanpei Chen, Tianhao Wu, Shengjie Wang, Xidong Feng, Jiechuan Jiang, Zongqing Lu, Stephen McAleer, Hao Dong, Song-Chun Zhu, and Yaodong Yang. Towards human-level bimanual dexterous manipulation with reinforcement learning. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 5150–5163. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/217a2a387f52c30755c37b0a73430291-Paper-Datasets_and_Benchmarks.pdf.

Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. *Advances in neural information processing systems*, 30, 2017.

Yuqing Du, Olivia Watkins, Zihan Wang, Cédric Colas, Trevor Darrell, Pieter Abbeel, Abhishek Gupta, and Jacob Andreas. Guiding pretraining in reinforcement learning with large language models. In *International Conference on Machine Learning*, pp. 8657–8677. PMLR, 2023.

Íñigo Elguea-Aguinaco, Antonio Serrano-Muñoz, Dimitrios Chrysostomou, Ibai Inziarte-Hidalgo, Simon Bøgh, and Nestor Arana-Arexolaleiba. A review on reinforcement learning for contact-rich robotic manipulation tasks. *Robotics and Computer-Integrated Manufacturing*, 81:102517, 2023.

Jonas Eschmann. *Reward Function Design in Reinforcement Learning*, pp. 25–33. Springer International Publishing, Cham, 2021. ISBN 978-3-030-41188-6. doi: 10.1007/978-3-030-41188-6_3. URL https://doi.org/10.1007/978-3-030-41188-6_3.

Haozhi Fan and Jiawei Du. Forging better rewards: A multi-agent LLM framework for automated reward evolution, 2025. URL https://openreview.net/forum?id=Z6GStCfccl.

Jakob Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. Counterfactual multi-agent policy gradients. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

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

Adrian Goldwaser and Michael Thielscher. Deep reinforcement learning for general game playing. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pp. 1701–1708, 2020.

Qi Guo, Xing Liu, Jianjiang Hui, Zhengxiong Liu, and Panfeng Huang. Utilizing large language models for robot skill reward shaping in reinforcement learning. In *International Conference on Intelligent Robotics and Applications*, pp. 3–17. Springer, 2024.

Joshua Hare. Dealing with sparse rewards in reinforcement learning, 2019. URL https://arxiv.org/abs/1910.09281.

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

Minae Kwon, Sang Michael Xie, Kalesha Bullard, and Dorsa Sadigh. Reward design with language models. *arXiv preprint arXiv:2303.00001*, 2023.

Richard L Lewis, Satinder Singh, and Andrew G Barto. Where do rewards come from. In *Proceedings of the international symposium on AI-inspired biology*, pp. 2601–2606, 2010.

Hao Li, Xue Yang, Zhaokai Wang, Xizhou Zhu, Jie Zhou, Yu Qiao, Xiaogang Wang, Hongsheng Li, Lewei Lu, and Jifeng Dai. Auto mc-reward: Automated dense reward design with large language models for minecraft. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 16426–16435, 2024.

Lihe Li, Lei Yuan, Pengsen Liu, Tao Jiang, and Yang Yu. Llm-assisted semantically diverse teammate generation for efficient multi-agent coordination. In *Forty-second International Conference on Machine Learning*, 2025a.

Pengyi Li, HAO Jianye, 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 *Forty-second International Conference on Machine Learning*, 2025b.

Pengyi Li, Hongyao Tang, Yifu Yuan, and Jianye HAO. ReMAC: Large language model-driven reward design for multi-agent manipulation collaboration. In *Workshop on Scaling Environments for Agents*, 2025c. URL https://openreview.net/forum?id=CWYWhLho0a.

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

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

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, 2021. URL https://arxiv.org/abs/2108.10470.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35: 27730–27744, 2022.

Martin L Puterman. Markov decision processes. *Handbooks in operations research and management science*, 2:331–434, 1990.

Ilija Radosavovic, Tete Xiao, Bike Zhang, Trevor Darrell, Jitendra Malik, and Koushil Sreenath. Real-world humanoid locomotion with reinforcement learning. *Science Robotics*, 9(89):eadi9579, 2024.

Tabish Rashid, Mikayel Samvelyan, Christian Schroeder de Witt, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. Monotonic value function factorisation for deep multi-agent reinforcement learning. *Journal of Machine Learning Research*, 21(178):1–51, 2020. URL http://jmlr.org/papers/v21/20-081.html.

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

Christopher Stanton and Jeff Clune. Deep curiosity search: Intra-life exploration improves performance on challenging deep reinforcement learning problems. *arXiv preprint arXiv:1806.00553*, 2018.

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, 2025a.

Shengjie Sun, Jiafei Lyu, Runze Liu, Mengbei Yan, Bo Liu, Deheng Ye, and Xiu Li. Prof: An llm-based reward code preference optimization framework for offline imitation learning. *arXiv preprint arXiv:2511.13765*, 2025b.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, USA, 1998. ISBN 0-262-19398-1. URL http://www.cs.ualberta.ca/%7Esutton/book/ebook/the-book.html.

Dixiao Wei, Peng Yi, Jinlong Lei, Yiguang Hong, Hairong Dong, and Yuchuan Du. An automated reinforcement learning reward design framework with large language model for cooperative platoon coordination. *IEEE Transactions on Intelligent Transportation Systems*, 2026.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Yue Wu, Yewen Fan, Paul Pu Liang, Amos Azaria, Yuanzhi Li, and Tom M Mitchell. Read and reap the rewards: Learning to play atari with the help of instruction manuals. *Advances in Neural Information Processing Systems*, 36:1009–1023, 2023.

Tianbao Xie, Siheng Zhao, Chen 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*, volume 2024, pp. 35663–35699, 2024.

Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, et al. Deepseek-v4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026.

Hu Yifan, Hu Bin-Bin, Yuan Bowen, and Zhang Hai-Tao. Llm coach: Reward shaping for reinforcement learning-based navigation agent. In *2025 Joint International Conference on Automation-Intelligence-Safety (ICAIS) & International Symposium on Autonomous Systems (ISAS)*, pp. 1–6. IEEE, 2025.

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

Xuan Zhang, Chao Du, Tianyu Pang, Qian Liu, Wei Gao, and Min Lin. Chain of preference optimization: Improving chain-of-thought reasoning in llms. *Advances in Neural Information Processing Systems*, 37:333–356, 2024.

Kai Zhu and Tao Zhang. Deep reinforcement learning based mobile robot navigation: A review. *Tsinghua Science and Technology*, 26(5):674–691, 2021.

A Weight Optimization Details

This section provides the mathematical formulation of the weight optimization procedure described in the main paper. The full pipeline consists of three stages: credit assignment, credit normalization and fusion, and module selection. Here credit is a weighting signal for reward construction over additive reward modules, rather than an attribution of a shared team return to individual agents as in multi-agent credit assignment (Rashid et al., 2020; Foerster et al., 2018).

A.1 Credit Assignment

MLREF maintains two credits per module: LLM credit and correlation credit.

A.1.1 LLM Credit

After pool initialization or improvement, the LLM is prompted to select modules and propose initial weights. The assigned weights are normalized to sum to 1, serving as the LLM credit c_k^{LLM} for the current iteration; unselected modules receive zero credit.

A.1.2 Correlation Credit

During RL training in the previous iteration, let the performance curve be $P = [p_1, p_2, \ldots, p_T]$ and the averaged reward sequence per epoch of module m_k be $R = [r_1, r_2, \ldots, r_T]$, where T is the number of training epochs. Both sequences are aligned by epoch index: p_t is the value of the built-in task-specific success metric at epoch t, and r_t is the average reward of module m_k over epoch t.

Smoothing. To suppress noise, we apply moving-average smoothing with window size w_s to both sequences:

$$P_t^{(s)} = \frac{1}{w_s} \sum_{i=0}^{w_s-1} p_{t+i}, \qquad R_t^{(s)} = \frac{1}{w_s} \sum_{i=0}^{w_s-1} r_{t+i}, \tag{2}$$

for $t = 1, 2, \ldots, T - w_s + 1$.

Slow difference. To capture trends, we compute w_d-step differences of the smoothed sequences:

$$\Delta P_t = P_{t+w_d}^{(s)} - P_t^{(s)}, \qquad \Delta R_t = R_{t+w_d}^{(s)} - R_t^{(s)}, \tag{3}$$

for $t = 1, 2, \ldots, T - w_s - w_d + 1$.

Lagged correlation. To account for delayed effects of rewards on performance, we compute Pearson correlation at multiple lag steps of up to L epochs: $\ell = 0, 1, \ldots, L$.

$$\mathrm{corr}_\ell(X, Y) = \frac{\mathrm{Cov}\big(X_{\ell+1:T'},\ Y_{1:T'-\ell}\big)}{\sigma\big(X_{\ell+1:T'}\big) \cdot \sigma\big(Y_{1:T'-\ell}\big)}, \tag{4}$$

where (X, Y) can be either the smoothed pair $(P^{(s)}, R^{(s)})$ or the differenced pair $(\Delta P, \Delta R)$, T' is the sequence length after smoothing/differencing, $\mathrm{Cov}(\cdot, \cdot)$ denotes covariance, and $\sigma(\cdot)$ denotes standard deviation.

The raw and differential correlation credits for module m_k are:

$$c_k^{\text{raw}} = \max_{0 \le \ell \le L} \text{corr}_\ell\big(P^{(s)},\ R^{(s)}\big),$$
$$c_k^{\text{diff}} = \max_{0 \le \ell \le L} \text{corr}_\ell\big(\Delta P,\ \Delta R\big). \tag{5}$$

To avoid dividing by a near-zero denominator, we skip any lag at which either standard deviation in Eq. 4 falls below a small threshold ϵ, and set the corresponding credit to zero if all lags are skipped.

The final correlation credit is a weighted combination:

$$c_k^{\text{corr}} = w_{\text{raw}} \cdot c_k^{\text{raw}} + (1 - w_{\text{raw}}) \cdot c_k^{\text{diff}}, \tag{6}$$

where $w_{\text{raw}} \in [0, 1]$ balances the contribution of raw versus differential correlation.

A.1.3 EMA Smoothing

To prevent credit values from oscillating across iterations, both LLM and correlation credits are smoothed via exponential moving average with update rate α:

$$\tilde{c}_k^{\text{LLM}} \leftarrow \alpha \cdot c_k^{\text{LLM}} + (1 - \alpha) \cdot \tilde{c}_k^{\text{LLM}},$$
$$\tilde{c}_k^{\text{corr}} \leftarrow \alpha \cdot c_k^{\text{corr}} + (1 - \alpha) \cdot \tilde{c}_k^{\text{corr}}. \tag{7}$$

Both credits are initialized to 0 and are retained for modules modified in place, whereas an added or rewritten module starts from zero. Throughout, c_k denotes the raw credit from the current iteration and $\tilde{c}_k$ denotes its exponentially smoothed estimate. A larger α makes the credit more responsive to current observations; a smaller α yields smoother evolution.

A.2 Credit Normalization and Fusion

Before fusion, the two smoothed credits $\tilde{c}_i^{\text{LLM}}$ and $\tilde{c}_i^{\text{corr}}$ are normalized to comparable ranges. Let M be the number of modules in the current pool variant.

LLM credit is non-negative and normalized linearly:

$$s_i^{\text{llm}} = \frac{\tilde{c}_i^{\text{LLM}}}{\sum_{j=1}^{M} \tilde{c}_j^{\text{LLM}}}, \tag{8}$$

with a uniform distribution $1/M$ applied if all $\tilde{c}_j^{\text{LLM}} = 0$.

Correlation credit $\tilde{c}_i^{\text{corr}} \in [-1, 1]$ is normalized via softmax with temperature τ, subtracting the maximum for numerical stability:

$$s_i^{\text{corr}} = \frac{\exp\big((\tilde{c}_i^{\text{corr}} - \max_k(\tilde{c}_k^{\text{corr}}))\,/\,\tau\big)}{\sum_{j=1}^{M} \exp\big((\tilde{c}_j^{\text{corr}} - \max_k(\tilde{c}_k^{\text{corr}}))\,/\,\tau\big)}. \tag{9}$$

Fusion. The normalized credits are fused into a composite weight:

$$s_i = w_{\text{LLM}} \cdot s_i^{\text{llm}} + (1 - w_{\text{LLM}}) \cdot s_i^{\text{corr}}, \tag{10}$$

where $w_{\text{LLM}} \in [0, 1]$ controls the relative weight of LLM judgment versus empirical evidence.

A.3 Module Selection with UCB

To balance exploitation and exploration, an upper confidence bound (UCB) bonus (Auer et al., 2002) is added to the composite weight:

Parameter	Value
MLREF — Optimization	
Max iterations I	9
Parallel samples per iteration N	3
Pipeline runs	3
RL evaluation seeds	5
MLREF — Pool Merge	
Soft update threshold t	0.1
MLREF — Correlation Credit	
Smoothing window size w_s	30
Differential window size w_d	30
Max lag steps L	10
Correlation threshold ϵ	10^{-6}
Raw correlation weight w_{raw}	0.5
MLREF — Credit Fusion	
EMA update rate α	0.7
Softmax temperature τ	0.25
LLM credit weight w_{LLM}	0.5
MLREF — Module Selection	
UCB exploration rate γ	0.2
Max modules selected K	5
RL Training	
Algorithm	PPO
PPO hyperparameters	Same as EUREKA (Ma et al., 2024)
Environment epochs (iterative optimization)	3,000
Environment epochs (final evaluation)	20,000
LLM Configuration	
Model	DeepSeek-V4-Flash
Mode	Reasoning (thinking mode)
Baselines (following original configurations)	
EUREKA (Ma et al., 2024) — Iterations	5
EUREKA (Ma et al., 2024) — Samples per iteration	16
RF-Agent (Gao et al., 2026) — MCTS updates	80

Table 3: Hyperparameters for MLREF and baseline methods.

$$UCB_i = s_i + \gamma \cdot \frac{\sqrt{\ln(U+1)}}{u_i + 1}, \tag{11}$$

where s_i is the fused composite weight, U is the total number of times any module from the current pool variant has been selected across iterations, u_i is the selection count of module m_i, and γ is the UCB exploration coefficient. The top-K modules ranked by UCB_i are selected, and their composite weights s_i are used directly in the reward combination (Eq. 1).

B HYPERPARAMETERS

Table 3 lists all hyperparameters used in MLREF and the baseline methods.

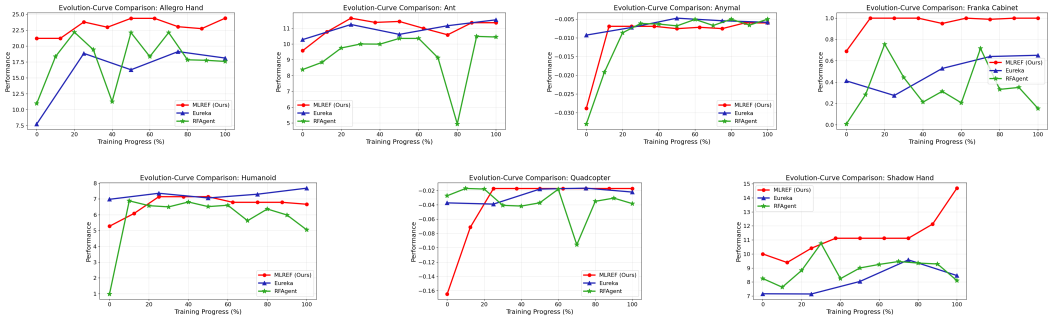

Figure 3: Evolution curves for all 7 Isaac Gym tasks.

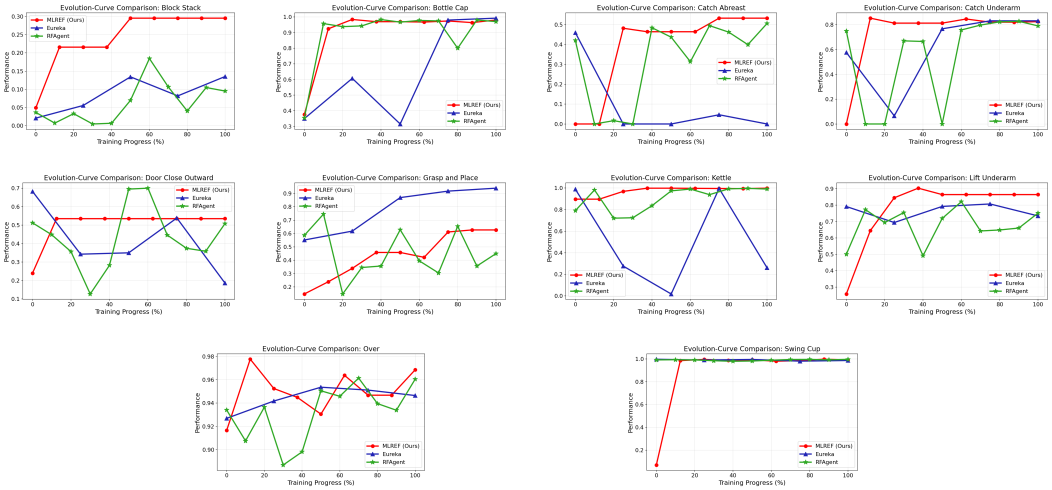

Figure 4: Evolution curves for all 10 Bi-DexHands tasks.

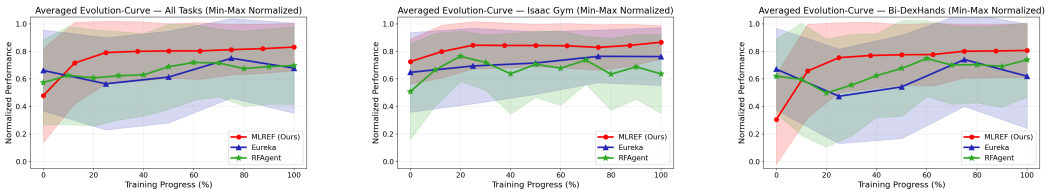

Figure 5: Average evolution curves. Left: all 17 tasks. Middle: the 7 Isaac Gym tasks. Right: the 10 Bi-DexHands tasks. Per-task curves are min-max normalized before averaging.

C Iterative Evolution Curves

This section presents the performance evolution across optimization iterations for all 17 tasks and three suite-level averages. For per-task curves, solid lines denote the best sample for each iteration; for average curves, solid lines and shaded areas represent the mean and standard deviation over tasks. Per-task curves are min-max normalized before averaging to compute suite-level curves.

Isaac Gym suite. Figure 3 shows evolution curves for all 7 Isaac Gym tasks (4 locomotion and 3 dexterous manipulation).

Bi-DexHands suite. Figure 4 shows evolution curves for all 10 Bi-DexHands tasks.

Suite-level averages. Figure 5 reports the average evolution curves.

D LLM PROMPT TEMPLATES

This section presents the prompts used to invoke the LLM at key stages of MLREF. Content enclosed in curly braces {} denotes dynamic fields populated at runtime. Unless otherwise noted, all prompts share the following system-level preamble.

D.1 SYSTEM PROMPT

```
You are a reward engineer trying to write reward functions to solve
    reinforcement learning tasks as effective as possible.
Your goal is to write a reward function for the environment that will
    help the agent learn the task described in text.
Your reward function should use useful variables from the environment
    as inputs. As an example, the reward function signature can be: {
    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.
```

D.2 JSON OUTPUT FORMAT CONSTRAINT

All prompts that request structured output from the LLM append the following format specification. Individual prompts below omit this block for brevity.

```
You are allowed to reason internally before producing the final answer
    .
Your final output must be a single JSON object that strictly follows
    the schema below:
{schema}
Do not include your reasoning or any extra text.
```

D.3 INITIAL REFLECTION PROMPTS

These prompts are issued once before the first iteration to analyze the task and environment.

D.3.1 TASK REFLECTION

```
Before designing reward functions, briefly reflect on the task to
    understand the goal.
Focus on the following:
1. What is the final success condition of the task? Describe only the
    end result that indicates success.
2. What are some possible behaviors (including non-intuitive or
    unexpected ones) that could lead to success?
3. What are some common failure patterns where the agent appears to
    make progress but never actually completes the task?

Important guidelines:
- Do NOT assume the task must be solved in well-defined stages or
    steps.
- Do NOT assume the behavior needs to be smooth, stable, or human-like
    .
- Success may arise from aggressive, unstable, or surprising
    interactions.
- Avoid over-structuring the problem; keep the reasoning flexible and
    open-ended.

Keep the reflection concise and avoid over-decomposition.
Do NOT design any reward code at this stage. The refined reward code
    will be designed in the next stage based on your reflections here.
```

D.3.2 ENVIRONMENT REFLECTION

```
Before designing the reward functions, let's reflect on the following
    question to better understand the environment:

Question: What are the relevant state variables available in the
    environment that can be used to design the reward functions?
    Figure out their types and usages. Also identify their shapes if
    they are tensors.

Note that the available variables are those defined in the class with
    prefix 'self.' in the environment code.
Do not use the input variables of the functions, as they may be name
    aliases and not directly accessible in the reward code.

Please include as many variables as possible, since the reward code
    will be based on these variables.

Do NOT design any reward code at this stage. The refined reward code
    will be designed in the next stage based on your reflections here.
```

D.4 POOL INITIALIZATION PROMPTS

These prompts construct the initial module pool from scratch. Generation proceeds in two steps: first specifications, then implementations.

D.4.1 SPECIFICATION GENERATION

```
Design some reward modules that will be included in the module pool.
Just provide the function specifications and a brief description of
    what each module does.
Each module should be named as "<aspect>_reward" for clarity.
Generate 4 to 6 different reward modules.

Tips for designing modules:
(1) Consider various reward shaping techniques, such as distance-based
     rewards, progress-based rewards, and task-specific rewards.
(2) Consider make use of every relevant state information available in
     the environment to design informative reward modules.
(3) Ensure that the modules are diverse and capture different aspects
    of the task.
```

D.4.2 MODULE IMPLEMENTATION

```
Implement the python code for a reward module based on this
    specification:
{specification}

Rules for writing the reward module code:
(1) The module should only contain a single function that computes the
     reward. Do not include any helper functions.
(2) The return type of the function must be torch.Tensor.
(3) The function name and input variables MUST match the specification
    . Explicitly specify the type of each input variable and the
    return type of the function.
(4) The code output should be formatted as a python code string: "```
    python ... ```".
(5) The code will be run using TorchScript, so it should be compatible
     with TorchScript.

Some helpful tips for writing the reward function code:
(1) You may find it helpful to normalize the reward to a fixed range
    by applying transformations like torch.exp to the reward
```

```
(2) If you choose to transform a reward component, then you must also
    introduce a temperature parameter inside the transformation
    function; this parameter must be a named variable in the reward
    function and it must not be an input variable. Each transformed
    reward component should have its own temperature variable
```

D.5 Pool Improvement Prompt

From the second iteration onward, these prompts improve the module pool based on the training statistics of the previous iteration. The improvement proceeds in two steps: first generate a plan, then execute the plan. In particular, during the execution step, the LLM uses the module implementation prompt above for module addition and rewrite, while module deletion is handled without invoking the LLM. Thus only the module modification prompt is presented for the execution step.

D.5.1 Improvement Plan Generation

```
Based on the training statistics of the RL training, please fix the
    bugs in the existing modules or suggest improvements to the
    current module pool to better suit the task requirements.
The details of the current module pool are as follows:
{module_pool_details}

You can select from the following types of actions to improve the
    module pool:
(1) ADD: Adding new modules to the pool. You should provide a full
    module specification for the new module without implementing the
    code.
(2) DELETE: Deleting modules that are redundant or not useful for the
    task.
(3) MODIFY: Making small changes to existing modules to better align
    with the task requirements. You should provide a natural language
    description of the required changes. The module specification
    CANNOT be changed in this way.
(4) REWRITE: Making big refactors to existing modules to better align
    with the task requirements. The module specification CAN be
    changed in this way, but the name of the module should remain the
    same for easier tracking.
For each action, please provide a clear natural language explanation
    of the reasoning behind it. You can refer to the training
    statistics if applicable.

Some helpful tips for analyzing the policy feedback:
(1) If the task score is always near zero, then you must consider
     (a) Deleting modules that may mislead the agent.
     (b) Adding new modules that better capture important aspects of
         the task.
(2) If the values for a certain reward component are near identical
    throughout, then this means RL is not able to optimize this
    component as it is written. You may consider
     (a) Changing its scale or the value of its temperature parameter.
     (b) Deleting it from the pool.
(3) If some reward components' magnitude is significantly larger, then
     you must re-scale its value to a proper range.
(4) The 'compute_reward' function is automatically generated based on
    the modules and the assembly plan, so you should not directly
    modify the code of 'compute_reward'. Instead, you should modify
    the modules in the pool and let the system automatically generate
    the new 'compute_reward' function based on the updated modules and
     assembly plan.
```

D.5.2 Module Modification

```
Modify the code of the following module based on the description of
    required changes.

Module specification:
{spec}

Current module code:
{code}

Required changes:
{description}

Do not change the module specification. Only modify the code as per
    the required changes.
Rules for writing the reward module code:
(1) The module should only contain a single function that computes the
     reward. Do not include any helper functions.
(2) The return type of the function must be torch.Tensor.
(3) The function name and input variables MUST match the specification
    . Explicitly specify the type of each input variable and the
    return type of the function.
(4) The code output should be formatted as a python code string: "```
    python ... ```".
(5) The code will be run using TorchScript, so it should be compatible
     with TorchScript.
```

D.6 Feedback Reflection Prompts

The feedback reflection prompt varies depending on the outcome of the previous RL training iteration. We present the three representative variants below.

D.6.1 Training Failed

```
The refined reward function in the CURRENT iteration did not run
    successfully, so we roll back to the previous BEST module pool
    design as the basis of our reflection and improvement.

Here is the previous BEST module pool for your reference:
{module_pool_details}

Here is the module pool update plan and the error message in the
    CURRENT iteration:
{improve_plan}
{training_signal}

{env_plugin}

Before designing the new reward functions based on the BEST module
    pool again, let's reflect on the following questions to better
    understand the error and identify potential issues in the reward
    function design:
1. What is the critical error message in the training signal?
2. Where is the error occurring in the code?
3. Based on the provided environment, what could be the potential
    reasons for this error?
4. How can we avoid this error in the next iteration of module pool
    improvement?

Please think step by step and provide your reflections on these
    questions.
Just answer the questions one by one using natural language, and feel
    free to provide any additional insights or observations that may
    be relevant.
```

```
Do NOT design any reward code at this stage. The refined reward
    function will be designed in the next stage based on your
    reflections here.
```

Here "env_plugin" contains the task description and the environment code, along with the task reflection and environment reflection if available.

D.6.2 TRAINING SUCCEEDED BUT DID NOT SURPASS BEST

```
The performance of the CURRENT module pool design has regressed
    compared to the previously BEST performance, so we roll back to
    the BEST module pool as the basis of our reflection and
    improvement.

Here is the BEST module pool for your reference:
{module_pool_details}

Here is the BEST module assembly along with the training statistics
    from the BEST iteration:
{best_module_usage_list}
{best_training_signal}

Here is the module pool update plan in the CURRENT iteration:
{improve_plan}

Here is the module assembly and statistics from the CURRENT iteration
    for comparison:
{module_usage_list}
{training_signal}

Before designing new improvement plan based on the BEST module pool
    again, let's reflect on the following questions to better
    understand the statistics and identify potential issues in the
    current module pool design:
1. What are the key trends and overall performance observed in the
    training statistics?
2. What are the differences in the module design and assembly between
    the current and best iterations? Are they responsible for the
    observed performance regression?
3. How does each reward module contribute to the overall performance?
    Are there any modules that seem to be more effective or less
    effective based on the statistics? Provide some suggestions on the
     future use of these modules.
4. Are there any potential issues or limitations in the current reward
     function design? How can we address these issues in the next
    iteration?

Please think step by step and provide your reflections on these
    questions.
Just answer the questions one by one using natural language, and feel
    free to provide any additional insights or observations that may
    be relevant.

Notice that we have rolled back to the BEST module pool, so the next
    step improvement plan will be based on the BEST module pool.
Try to come up with some new ideas to avoid the repeated regression.
Do NOT generate any specific plan at this stage. The pool improvement
    plan will be designed in the next stage based on your reflections
    here.
```

D.6.3 TRAINING SUCCEEDED AND SURPASSED BEST

```
Here is the current module pool details for your reference:
{module_pool_details}
```

```
Here is the best module assembly along with the training statistics
    from the CURRENT iteration:
{module_usage_list}
{training_signal}

Here is the best module assembly and statistics from the PREVIOUS
    iteration for comparison:
{module_usage_list_prev}
{training_signal_prev}

Here is the module pool update plan in the PREVIOUS iteration:
{improve_plan}

Before designing new improvement plan for the module pool, let's
    reflect on the following questions to better understand the
    statistics and identify potential issues in the current module
    pool design:
1. What are the key trends and overall performance observed in the
    training statistics? Are there any significant improvements or
    regressions compared to the previous iteration?
2. What are the differences in the module design and assembly between
    the current and previous iterations? Are they responsible for the
    performance regression?
3. How does each reward module contribute to the overall performance?
    Are there any modules that seem to be more effective or less
    effective based on the statistics? Provide some suggestions on the
     future use of these modules.
4. Are there any potential issues or limitations in the current reward
     function design? How can we address these issues in the next
    iteration?

Please think step by step and provide your reflections on these
    questions.
Just answer the questions one by one using natural language, and feel
    free to provide any additional insights or observations that may
    be relevant.
Do NOT generate any specific plan at this stage. The pool improvement
    plan will be designed in the next stage based on your reflections
    here.
```

D.7 Weight Selection Prompt

These prompts ask the LLM to select modules and assign weights, producing the LLM credit used in hybrid credit assignment. The templates vary between the initialization phase and the iterative optimization phase.

D.7.1 Weight Selection Prompt for Initialization Phase

```
Here is the created module pool, along with the specifications of each
     module.
{module_pool}

You should construct the reward function as a linear combination of
    the modules in the pool.
Please choose the most appropriate modules and decide their weights to
     construct the reward function for the given task.

Here are some tips for choosing weights:
(1) The weights should be non-negative, as all modules are designed to
     provide positive feedback for desirable behaviors.
(2) The weights should have balanced magnitudes to ensure that no
    single module dominates the reward signal.
(3) Consider the relative importance of each module in achieving the
    task objectives when assigning weights.
```

D.7.2 WEIGHT SELECTION PROMPT FOR ITERATIVE OPTIMIZATION PHASE

```
Here is the modified module pool, along with the specifications of
    each module.
{module_pool}

You should construct the reward function as a linear combination of
    the modules in the modified pool.
Please choose the most appropriate modules and decide their weights to
     construct the reward function for the given task.
You can also refer to the training statistics of the previous RL
    training and the improvement plan to help you find the most
    promising modules.

Tips for selecting modules:
Consider including these modules in the final reward function:
(1) The modules whose reward values changed a lot during training, as
    they are likely to be more effective for RL optimization.
(2) The modules that were modified in the improvement plan, as they
    are expected to better align with the task requirements.
(3) The newly added modules in the improvement plan, as they may
    capture important aspects of the task that were previously missing
    .

Tips for choosing weights:
(1) The weights should be non-negative, as all modules are designed to
     provide positive feedback for desirable behaviors.
(2) The weights should have balanced magnitudes to ensure that no
    single module dominates the reward signal.
(3) Consider the relative importance of each module in achieving the
    task objectives when assigning weights.
```

The template only includes the current module pool for reference, as the previous RL training statistics and the improvement plan have been provided in the chat context.