Uncertainty-aware Reward Design Process

Anonymous authors
Paper under double-blind review

Abstract

Designing effective reward functions is a cornerstone of reinforcement learning (RL), yet it remains a challenging process due to the inefficiencies and inconsistencies inherent in conventional reward engineering methodologies. Recent advances have explored leveraging large language models (LLMs) to automate reward function design. However, their suboptimal performance in numerical optimization often yields unsatisfactory reward quality, while the evolutionary search paradigm demonstrates inefficient utilization of simulation resources, resulting in prohibitively lengthy design cycles with disproportionate computational overhead. To address these challenges, we propose the Uncertainty-aware Reward Design Process (*URDP*), a novel framework that integrates large language models to streamline reward function design and evaluation in RL environments. *URDP* quantifies candidate reward function uncertainty based on the self-consistency analysis, enabling simulation-free identification of ineffective reward components while discovering novel reward components. Furthermore, we introduce uncertainty-aware Bayesian optimization (UABO), which incorporates uncertainty estimation to significantly enhance hyperparameter configuration efficiency. Finally, we construct a bi-level optimization architecture by decoupling the reward component optimization and the hyperparameter tuning. *URDP* orchestrates synergistic collaboration between the reward logic reasoning of the LLMs and the numerical optimization strengths of the Bayesian Optimization. We conduct a comprehensive evaluation of *URDP* across 35 diverse tasks spanning three benchmark environments: IsaacGym, Bidexterous Manipulation, and ManiSkill2. Our experimental results demonstrate that *URDP* not only generates higher-quality reward functions but also achieves significant improvements in the efficiency of automated reward design compared to existing approaches.

1 Introduction

In reinforcement learning (RL), the design of reward functions serves as a pivotal determinant for successfully training agents in sequential decision-making tasks. These rewards guide the learning process by shaping agent behaviors to accomplish complex objectives across diverse environments. While conventional approaches such as reward engineering and inverse reinforcement learning (IRL) Arora & Doshi (2021) established early research paradigms, they remain fundamentally constrained by their reliance on human expertise and the availability of high-quality demonstration data, particularly in domains like robotic skill acquisition Zitkovich et al. (2023). Recent advancements in large language models (LLMs) Radford et al. (2019); Brown et al. (2020); Liu et al. (2024) have demonstrated remarkable capabilities in natural language understanding, code generation, and contextual optimization, thereby introducing a novel paradigm for automated reward function design.

However, current automated reward design methodologies based on LLMs present two fundamental challenges Cao et al. (2024). First, the **efficiency** of reward function design remains suboptimal. Existing approaches rely heavily on simulation-based training processes Ma et al. (2024a) to evaluate reward function efficacy, which involves extensive and often redundant evaluations, leading to significant computational overhead without commensurate benefits. Second, the **performance** of LLM-generated reward functions frequently falls short of expectations. During the optimization process, LLMs fail to fully leverage their reasoning capabilities, resulting in reward functions that inadequately capture the intended task objectives.

Given that the design efficiency and generation quality of reward functions are closely tied to the speed and effectiveness of policy learning, a key research question emerges: *How can we enhance the efficiency of obtaining high-quality reward functions?*

In this paper, we introduce the **Uncertainty-aware Reward Design Process** (*URDP*), a novel framework for automated reward function generation. First, we propose a method to quantify the uncertainty of generated samples, which enables the selective elimination of redundant reward function sampling and simulation. This approach is grounded in a key observation regarding self-consistency Wang et al. (2022): LLMs exhibit higher output consistency when handling well-defined tasks, allowing for more efficient sampling strategies. Second, we identify a critical limitation in current LLM-based evolutionary search approaches, i.e., their suboptimal performance in numerical optimization. To address this, we decouple reward component formulation from reward intensity optimization, delegating the latter to a dedicated numerical optimization module. Specifically, we propose a novel Bayesian optimization Snoek et al. (2012) approach incorporating uncertainty distribution priors, which significantly accelerates convergence in this black-box optimization task. Experimental results demonstrate that *URDP* surpasses state-of-the-art methods in both reward quality and computational efficiency, establishing a new benchmark for automated reward design.

Overall, our contributions are summarized as follows: (1) We propose *URDP*, a novel framework for automated reward function design in reinforcement learning. The framework employs an alternating bi-level optimization process that decouples reward component design from hyperparameter optimization, thereby significantly enhancing reward function performance. (2) We introduce a self-consistency-based reward uncertainty quantification method for the reward component design process. This approach not only dramatically improves the efficiency of reward component validation but also facilitates the discovery of novel reward logic. (3) We present an Uncertainty-aware Bayesian optimization algorithm that substantially increases the efficiency of hyperparameter search. (4) Our comprehensive evaluation across 35 tasks spanning 3 distinct benchmarks demonstrates that *URDP* consistently outperforms existing methods in both reward function generation efficiency and final policy performance, as evidenced by rigorous quantitative analysis.

2 Related work

Reward code generation. The automation of reward code generation has been a critical area of research, aiming to simplify and improve the process of defining task-specific reward functions for RL. As the dual formulation of RL problems, inverse reinforcement learning (IRL) methods Arora & Doshi (2021) have been extensively investigated for reward function acquisition. However, these approaches are fundamentally limited by their dependence on demonstration data, which severely constrains their scalability in practical applications. Recently, LLM-based reward design methodologies Kwon & Michael (2023); Yu et al. (2023); Ma et al. (2024a); Xie et al. (2024) have demonstrated promising potential, offering a paradigm shift in automated reward function development. L2R Yu et al. (2023) introduced a two-stage LLM-prompting framework to generate templated rewards, bridging high-level language instructions with low-level robot actions. Eureka Ma et al. (2024a) leveraged the zero-shot and in-context learning capabilities of advanced LLMs to perform evolutionary optimization over reward code. This method demonstrated the potential of LLMs to generate rewards without task-specific prompting or predefined templates, enabling agents to acquire complex skills via RL. Text2Reward Xie et al. (2024) extended this line of work by generating shaped, dense reward functions as executable programs grounded in compact environment representations. Unlike sparse reward codes or constant reward functions, Text2Reward produces interpretable, dense reward codes capable of iterative refinement with human feedback. Despite their success, these methods overlook two key limitations: (1) insufficient reasoning capability for reward logic derivation, and (2) inadequate exploration of novel reward components. Our proposed framework decouples confounding factors in reward design to reduce cognitive load and incorporates uncertainty quantification to guide LLMs toward more focused analysis and refinement of reward logic relevance.

Hybrid optimization. Recent advances in large language models (LLMs) OpenAI (2023); Liu et al. (2024) have demonstrated remarkable progress in text-based complex reasoning tasks Li et al. (2025). Through techniques such as self-improvement Song et al. (2023), multi-path reasoning Wan et al. (2024), and reward modeling Zhong et al. (2025), LLMs exhibit substantial potential in contextual comprehension OpenAI

(2023), code generation Yu et al. (2024), and task planning Hao et al. (2023). However, their capabilities in deep logical reasoning Cheng et al. (2025), particularly in mathematical and numerical optimization domains Yan et al. (2025), remain underexplored, with significant performance gaps persisting. Several studies attempt to employ LLMs as meta-optimizers for diverse optimization problems Yang et al.. Yet, due to their inherently discrete representation nature, LLMs' effectiveness in high-dimensional, continuous numerical reasoning tasks requires further investigation Assran et al. (2025). In reinforcement learning, the reward design inherently involves multiple types of optimization problems. Unlike completely LLM-based evolutionary search approaches Ma et al. (2024b); Xie et al. (2024), our work introduces black-box numerical optimization tools to compensate for the limitations of LLMs in continuous numerical optimization.

Uncertainty quantification. Uncertainty quantification (UQ) serves as a fundamental component for reliable automated decision-making and has been extensively studied in domains such as Bayesian inference Gal & Ghahramani (2015); Foong et al. (2020). Recent advances have investigated uncertainty quantification in black-box language models Liu et al. (2025); Geng et al. (2024); Kuhn et al. (2023); Lin et al., yielding various approaches including token-level entropy methods Kadavath et al. (2022), conformal prediction-based techniques Su et al. (2024), and consistency-based frameworks Lin et al.. While existing methods primarily leverage uncertainty estimation to enhance LLM interpretability Ahdritz et al. (2024) and mitigate hallucination risks Shorinwa et al. (2024); Mohri & Hashimoto (2024), our methodology not only actively quantifies prediction uncertainty in LLMs but also strategically leverages this uncertainty as the foundational mechanism for both simulation-free reward function design and efficient numerical optimization. Furthermore, we identify and characterize a significant correlation between reward component uncertainty and the discovery of novel reward formulations. This targeted uncertainty quantification framework represents a substantive methodological contribution that enables more effective and reliable reward design for reinforcement learning tasks.

3 Preliminary: problem setup and notations

Reinforcement learning (RL) tasks can be modeled as Markov Decision Processes (MDPs) defined by the tuple $\langle \mathcal{S}, \mathcal{A}, P, R, \gamma \rangle$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $P(s'|s,a)$ is the transition probability, $R(s,a)$ is the reward function, and γ is the discount factor.

Reward design problem (RDP) Singh et al. (2009). Despite its superior interpretability, designing rewards $R(s,a)$ represented in code form is critical to aligning agent behavior with task objectives but involves challenges such as managing redundancies, inconsistencies, and uncertainties during reward function generation. Following the previous works Song et al. (2023); Ma et al. (2024a); Xie et al. (2024), the reward (R) is represented as a function of the reward components ($\mathbf{r}$) and the reward intensity (θ)

$$R = f(\mathbf{r}, \theta). \tag{1}$$

Conventionally, evaluating the quality of a reward function candidate necessitates computationally expensive simulations (RL training). In this work, we aim to simultaneously maximize reward function performance while minimizing the associated simulation cost throughout the automatic reward design process.

Bayesian optimization (BO) is a sequential design strategy for global optimization of black-box functions Shahriari et al. (2015); Snoek et al. (2012). Given a black-box function $f : \mathbb{X} \to \mathbb{R}$, Bayesian optimization aims to find an input $\mathbf{x}^* \in argmin_{x \in \mathbb{X}} f(\mathbf{x})$ that globally minimizes f. It places a prior $p(f)$ over the objective function f to form a surrogate model (usually a Gaussian process Wang et al. (2024)). An *acquisition function* (e.g., the Expected Improvement (EI) Ament et al. (2023)) $a_{p(f)} : \mathbb{X} \to \mathbb{R}$ strategically determines the direction of the search for sampling points. The algorithm iteration proceeds in the following three steps: (1) find the most promising $\mathbf{x}_{n+1} \sim argmax_{a_p(\mathbf{x})}$; (2) evaluate the function $y_{n+1} \sim f(\mathbf{x}_{n+1}) + \mathcal{N}(0, \sigma^2)$ and update the set of historical observations $\mathcal{D}_n = (x_j, y_j)_{j=1...n}$ by adding the point $(\mathbf{x}_{n+1}, y_{n+1})$, and (3) update $p(f|\mathcal{D}_{n+1})$ and $a_{p(f|\mathcal{D}_{n+1})}$.

4 Methods

The Uncertainty-aware Reward Design Process (*URDP*) framework incorporates three fundamental elements: (1) Decoupling of the reward component and reward intensity design processes, (2) Reward component generation based on uncertainty quantification, and (3) Uncertainty-aware Bayesian optimization. Collectively, *URDP* enhances reward design efficiency by minimizing redundant simulations while improving the quality of generated reward functions through the integration of numerical optimization techniques within the decoupled optimization framework.

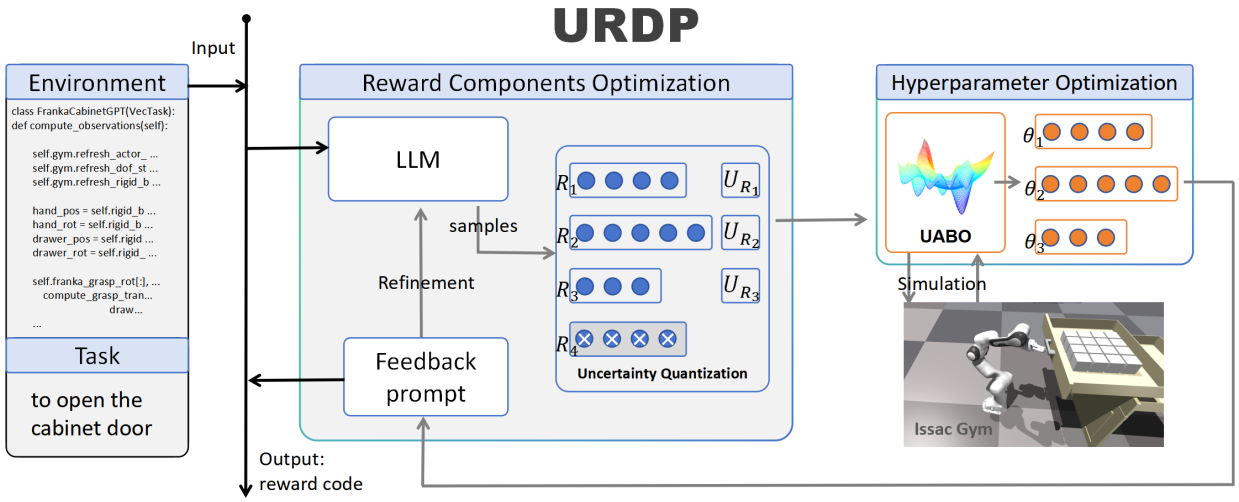


Figure 1: *URDP* implements an alternating bi-level iterative optimization framework for automated reward design problems (RDP). The outer-loop optimization employs LLMs to refine reward components, where uncertainty quantification significantly enhances sampling efficiency. Concurrently, the inner-loop optimization utilizes Uncertainty-Aware Bayesian Optimization (UABO) to determine optimal hyperparameter configurations for the reward components. The decoupled architecture strategically leverages the complementary strengths of LLMs in conceptual reward design and numerical optimization tools in precise parameter tuning, achieving synergistic improvements in both final policy performance and computational efficiency.

4.1 Decoupled Reward Generation and Hyperparameter Optimization

Large language models (LLMs) possess extensive commonsense knowledge about task rewards, enabling them to surpass human-level performance in designing reward components Yu et al. (2023); Ma et al. (2024a); Xie et al. (2024). However, their capability in black-box numerical optimization remains inferior to specialized numerical optimization algorithms (see Section 5.4 Abl-3 for an ablation study). Consequently, existing methods Ma et al. (2024a) that conflate the optimization of reward components with reward intensities not only yield suboptimal numerical optimization results but also lead to insufficient attention being paid to the optimization of reward components in agent learning.

URDP framework. Building upon these observations, we proposes a decoupled reward function design process. As illustrated in Figure 1, *URDP* implements a bi-level iterative optimization procedure where, given environment specifications and task descriptions, the agent first samples multiple reward functions from the LLM in the outer loop, ranks them using uncertainty quantification metrics, and filters out redundant and potentially unreliable rewards through a process termed *Reward Code Sampling with Uncertainty Screening*. Subsequently, the agent invokes numerical optimization tools in the inner loop to determine optimal reward intensity hyperparameters for the current reward configuration through black-box optimization and simulation-based evaluation. Finally, the agent evaluates the feasibility of current reward components and provides improvement feedback to refine the reward components. In essence, the outer loop optimizes the reward components and the reward logic while the inner loop tunes reward intensity hyperparameters, with their alternating optimization progressively converging toward optimal reward functions. See Alg. 1 for pseudocode.

Algorithm 1: Uncertainty-aware Reward Design Process

Input: Task description T, Environment code E, LLM $\mathcal{L}$.
Output: Optimized reward function R^*.
foreach *iteration* $n \in N_{outer}$ **do**
  Generate K reward component samples $\{r_{i,1}, r_{i,2}, \ldots, r_{i,m}\}_{i \in K}$ using $\mathcal{L}(T, E, prompt)$
  Uncertainty Quantization: $\{U(r_{i,1}), \ldots, U(r_{i,m})\}$ and $U(R_i)$
  Filter out redundancies and reserve $R_{i \in K^*}$
  foreach R_i, $t = 1, 2, \ldots N_{inner}/U(R_i)$ **do**
    Fit probabilistic model for $f(\{\theta_i\})$ on data $D_{i,t-1}$
    Choose $\{\theta_i\}_t$ by maximizing the acquisition function $uEI(\{\theta_i\}, \{U(r_i)\})$
    Evaluate by simulation training $y_{i,t} = f(\{\theta_i\}_t)$
    Augment the data $D_{i,t} = D_{i,t-1} \cup (\{\theta_i\}_t, y_{i,t})$
    Choose incumbent $\{\theta_i^*\} \leftarrow argmax\{y_{i,1}, \ldots, y_{i,t}\}$ and $y_i^* \leftarrow max\{y_{i,1}, \ldots, y_{i,t}\}$
  Refine *prompt* for the reward components
Choose optimal $\{(r^*, \theta^*)\} \leftarrow argmax\{y_1^*, \ldots, y_{K^*}^*\}$
Recombine $\{(r^*, \theta^*)\}$ into $R^*(s, a)$
return R^*.

4.2 Reward Code Sampling with Uncertainty Screening

In LLM-based RDP, automated reward function design can be achieved through iterative sampling and simulation. However, existing approaches indiscriminately conduct simulation-based evaluation on all LLM-generated reward function samples, despite the frequent presence of redundant or infeasible candidates within these samples. Our analysis identifies this as a critical factor contributing to computational inefficiency (see results in Section 5.4 Abl-1). Consequently, a pivotal question emerges: *how to effectively filter out potentially problematic reward function samples prior to simulations*, thereby avoiding computationally expensive yet futile simulation training.

Uncertainty priority. Our sampling approach is grounded in the principle of *self-consistency* Wang et al. (2022) in LLMs. When explicitly prompted to generate diverse outputs, LLMs that produce highly consistent responses demonstrate well-internalized, task-specific knowledge. Such outputs exhibit high reliability and typically require minimal refinement. Conversely, if the generated results show substantial diversity, this indicates uncertainty in the understanding of the task context and underlying concepts. These divergent results exhibit lower reliability and consequently demand more refinement.

Sampling and filtering. Based on this principle, the agent prompts the LLM to generate diverse reward components for a given RL task. The uncertainty score of each reward component r_i, denoted as $U(r_i)$, is quantified by its occurrence frequency across all sampled candidates. To quantify the uncertainty of the reward component, `URDP` identifies and resolves ambiguities in reward components using LLMs. It combines textual similarity and semantic similarity analyses to evaluate the relevance and clarity of reward components, assigning $U(r_i)$ to each reward component r_i as

$$U(r_i) = 1 - \sum_{i \in [1,K]} \left(u\left(max(S_{\text{text}}(r_i), S_{\text{semantic}}(r_i)) - \omega\right)\right)/K, \tag{2}$$

where $u(\cdot)$ is a step function, K denotes the quantity of the reward samples, ω is a decision parameter regarding the maximum similarity ($\omega = 0.95$), $S_{\text{text}} \in (0, 1]$ and $S_{\text{semantic}} \in (0, 1]$ are the textual and semantic similarity scores, respectively. Furthermore, the normalized sample uncertainty score $U(R_k)$ is computed to evaluate the overall uncertainty of each reward function sample R_k. Using $U(R_k)$, the agent filters out samples containing identical reward components, thereby eliminating redundant inner-loop optimization processes that would otherwise incur unnecessary computational overhead. Implementation details are elaborated in App. C.1. Moreover, our analysis reveals that highly uncertain reward components may contain unexplored components capable of facilitating effective reward shaping (see Section 5.5 Disc-2). Consequently, the agent implements an adaptive exploration-exploitation strategy. For high-uncertainty samples, it allocates additional inner-loop

iterations to prioritize exploration of optimal hyperparameter configurations for potentially novel reward components. For low-uncertainty samples, it emphasizes exploitation to minimize unnecessary simulations. This approach balances the trade-off between the exploration and utilization of uncertain reward components.

4.3 Uncertainty-aware Bayesian Optimization

While large language models demonstrate significant potential for reward component design, they exhibit suboptimal performance in numerical optimization tasks (see results in Section 5.4 Abl-3). This limitation leads to non-optimal reward intensity configurations in LLM-generated reward functions, representing a key factor in the poor policy learning performance observed in prior approaches. In contrast to existing methods, our *URDP* framework does not rely on LLM-based agents for direct numerical optimization. Instead, the agent serves as a controller that orchestrates specialized numerical optimization tools. Specifically, *URDP* delegates the inner-loop optimization of reward intensity parameters to Bayesian optimization (BO) algorithms. Benefiting from BO's superiority in black-box global optimization, this approach achieves significantly better performance than LLM-based optimization.

Although the classical Bayesian optimization algorithm, *i.e.*, Gaussian Process with Expected Improvement (EI) Ament et al. (2023); Snoek et al. (2012), demonstrates theoretical advantages, its practical efficiency remains unsatisfactory, particularly due to the substantial computational overhead incurred by acquisition functions during sampling (simulation training). This inefficiency frequently prevents convergence to globally optimal solutions within an acceptable number of samplings. Notably, the uncertainties $U(r_i)$ of individual reward components imply valuable prior knowledge for enhancing BO's sampling efficiency. Given a reward function comprising m components, we model the m reward intensities as a joint probability distribution. Crucially, higher uncertainty in r_i corresponds to a more uniform marginal distribution along that dimension. This observation suggests that sampling should prioritize exploitation over exploration in high-uncertainty dimensions. Building upon this smoothness assumption, we propose **Uncertainty-aware Bayesian optimization (UABO)** to address these limitations.

UABO incorporates reward component uncertainty scores, $U(r)$, into both the kernel function and acquisition function of the standard Bayesian optimization. The Matern kernel in Gaussian process has the form

$$k(p, p') = f_\nu(d) = \sigma^2 \cdot \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\frac{\sqrt{2\nu} d}{\ell} \right)^\nu K_\nu \left(\frac{\sqrt{2\nu} d}{\ell} \right), \tag{3}$$

where d is the Euclidean distance between p and p', σ^2 is the variance, ν is the smoothness parameter, ℓ is the length scale parameter and K_ν is the modified Bessel function of the second kind. We note that the kernel is isotropic, which means that all dimensions (i.e., the intensity parameters of reward components) share the same length scale parameter. To accommodate heterogeneous smoothness (uncertainty score $U(r_i)$) across different dimensions, we propose an anisotropic kernel function that incorporates uncertainty values as length scales within the distance metric. The distance is formulated as follows,

$$d_{\mathrm{u}}(p, p') = \sqrt{\left(\frac{x_1 - x_1'}{U(r_{i,1})} \right)^2 + \cdots + \left(\frac{x_n - x_n'}{U(r_{i,m})} \right)^2}. \tag{4}$$

Then the new kernel function is defined as

$$\tilde{k}(p, p') = f_\nu(d_{\mathrm{u}}) = \sigma^2 \cdot \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\frac{\sqrt{2\nu} d_{\mathrm{u}}}{U(R_i)} \right)^\nu K_\nu \left(\frac{\sqrt{2\nu} d_{\mathrm{u}}}{U(R_i)} \right). \tag{5}$$

Furthermore, we leverage $U(r)$ to enhance the performance of the acquisition function. The standard form of Expected Improvement (EI) acquisition function Ament et al. (2023) in Bayesian Optimization is as follows,

$$\mathrm{EI}_{y^\star}(x) = \mathbb{E}_{f(x) \sim \mathcal{N}(\mu(x), \sigma^2(x))} \left[[f(x) - y^\star]_+ \right] = \sigma(x) h \left(\frac{\mu(x) - y^\star}{\sigma(x)} \right), \tag{6}$$

where $[\cdot]_+ = \max(0\,,\,\cdot)$, $y^\star = \max_i y_i$ is the best observed value, and $h(z) = \phi(z) + z\Phi(z)$, ϕ is the standard normal distribution density and Φ is the distribution function.

To reduce inefficient exploration along directions with potentially insignificant influence on the function value, we introduce a penalty term that constrains the weighted distance between the candidate point and the current optimum, yielding an uncertainty-accelerated EI acquisition function (uEI) defined as

$$\mathrm{uEI}(\theta) = \mathrm{EI}(\theta) \cdot w(\theta), \tag{7}$$

$$w(\theta) = \exp\left(-\sum_{j=1}^{d} U(r_j)(\theta_j - \theta_j^\star)^2\right), \tag{8}$$

where θ denotes the reward intensity hyperparameter, $w(\theta)$ is a penalty term to constrain the weighted distance between θ and $\theta^\star$. Specifically, an uncertainty value approaching zero for a particular dimension indicates no restriction on variations along that dimension. Conversely, a large uncertainty weight in a certain direction implies that extensive exploration in that direction is discouraged. UABO demonstrates significantly improved convergence efficiency (see Section 5.4 Alb-3), reaching optimal values within limited hyperparameter search steps, thereby substantially enhancing the performance and effectiveness of reward intensity configuration. A formal proof of its convergence lower bound is provided in App. F.

5 Experiments

In this section, we evaluate the proposed *URDP* framework through extensive experiments on a diverse set of environments and tasks, comparing its performance against human and baseline approaches. All experiments and comparative analyses presented in this paper utilize DeepSeek-v3-241226 Liu et al. (2024) as the foundational model unless explicitly stated otherwise.

5.1 Baselines

Eureka Ma et al. (2024a) (Baseline) provides a systematic approach for generating reward functions utilizing LLMs. It incorporates feedback from various evaluation results to refine the generation of the reward function in evolutionary iterations. This iterative process continues until an optimal reward function is achieved.

Text2reward Xie et al. (2024) is a reinforcement learning method that automatically generates dense reward functions from natural language task descriptions using LLMs, without relying on expert data or demonstrations, and is able to express human goals in the form of procedural rewards, given to iterations using human feedback.

Human. To maintain a fair comparison, we adopted the same Human data as reported in Eureka Ma et al. (2024a). The original shaped reward functions provided in the benchmark tasks are developed by active reinforcement learning researchers who designed the tasks. These reward functions embody the outcomes of expert human reward engineering.

Sparse.These functions correspond to the fitness measures F employed to assess the quality of the generated reward signals. Analogous to human feedback, they are also provided as part of the benchmark suite. The detailed configurations for Dexterity tasks and Isaac tasks are consistent with those used in Eureka Ma et al. (2024a). The fitness functions of all tasks in ManiSkill2 are specified in App. B.2.

5.2 Experimental Setup

Benchmarks. Our environments consist of three benchmarks: Isaac, Dexterity and Maniskill2, and comprises 35 different tasks. Nine of these tasks are from the original Isaac Gym environment Nasir et al. (2024) (Issac), twenty are complex bi-manual tasks Chen et al. (2022) (Dexterity) and the remaining six are from the Maniskill2 environment Gu et al.. See App. B for more details.

Metrics. We examine fore metrics: i. Success Rate (**SR**). We report the success rates of different reward functions on Dexterity and ManiSkill2 tasks. To ensure fair comparison with the baseline methods, the

success rates for ManiSkill2 tasks are calculated using the last 50% of test results from each evaluation, while full test results are used for Dexterity tasks. ii.Human Normalized Score **(HNS)**. For Isaac tasks, following the evaluation setup in the Eureka, we employ the Human Normalized Score, $\frac{\text{Method-Sparse}}{|\text{Human-Sparse}|}$, as the evaluation metric. iii. The Number of Evaluations (**NOE**), quantified as the total number of simulations conducted across all samples during the optimization process. iiii. The Number of LLM Callings (**NLC**), representing the cumulative number of LLM calls made during both reward function generation and refinement. These metrics collectively provide comprehensive assessment, with SR and HNS evaluating the performance of the generated reward functions, and NOE and NLC quantifying design process efficiency.

Policy Learning. The performance of reward functions generated by *URDP* and comparative methods was rigorously validated through RL training. For both Isaac and Dexterity environments, we employ the same high-efficiency PPO Schulman et al. (2017) implementation as used in Eureka, using identical task-specific hyperparameters without modification. In the ManiSkill2 environment, we utilized both SAC Haarnoja et al. (2018) and PPO algorithms to ensure fair comparison between *URDP*, Text2Reward, and Eureka, strictly maintaining the original hyperparameter configurations across all methods. See App. C.2 for detailed parameter configurations.

5.3 Results

URDP improves the efficiency of the reward function design. Table 1 presents a comprehensive comparison of computational efficiency between *URDP* and Eureka across three benchmarks. When optimizing for peak reward performance, *URDP* requires only 52.4% of the simulation episodes (NOE) and 46.6% of the evolutionary iterations (NLC) compared to Eureka. This significant acceleration demonstrates the superior optimization efficiency of the *URDP* in automated reward function design. See App.E for a per-task breakdown.

Table 1: *URDP* demonstrates superior efficiency across all benchmarks.

	Isaac		Dexterity		ManiSkill2	
Methods	NOE↓	NLC↓	NOE↓	NLC↓	NOE↓	NLC↓
Txet2Reward	72.889	5	84.45	6.05	106.667	6.667
Eureka	68.667	4.556	80.05	5.5	98.667	6.167
URDP	**39.501**	**2.495**	**57.8**	**3.4**	**32.33**	**1.667**

URDP demonstrates superior reward function performance. Table 2 presents a systematic comparison of reward functions generated by different approaches across benchmark tasks. Under identical simulation budgets (NOE), reinforcement learning agents trained with *URDP*-derived reward functions achieve significantly higher success rates than competing methods. Notably, *URDP* demonstrates substantial performance gains of 132%, 45% and 76% over Eureka across the three experimental environments, representing substantial quality enhancements. Furthermore, *URDP*-designed reward functions outperform manually engineered counterparts by a considerable margin, providing compelling evidence for the efficacy of automated reinforcement learning frameworks.

Table 2: *URDP* exhibits higher reward quality across all benchmarks.

Methods	Isaac (HNS↑)	Dexterity (SR↑)	ManiSkill2 (SR↑)
Sparse	0	0.054	0.101
Human	1.000	0.459	0.434
Text2Reward	1.553	0.452	0.554
Eureka	1.607	0.466	0.449
URDP	**3.424**	**0.675**	**0.792**

URDP achieves synergistic progress in both generation quality and generation efficiency. In Figure 2, it can be intuitively seen that the superiority of *URDP* over Eureka, text2reward and Human, both in terms of the success rate and the reduction in the number of simulations, has been well improved on these

typical tasks. Each data point in the line plots represents the effectiveness of the reward function obtained after a single iteration. The results show that *URDP* surpasses human-designed rewards after just one LLM refinement cycle and achieves optimal performance with significantly fewer iterations than both Eureka and Text2Reward, which means that it consumes fewer tokens. These results collectively demonstrate that *URDP* achieves simultaneous improvements in both reward generation efficiency and design quality.

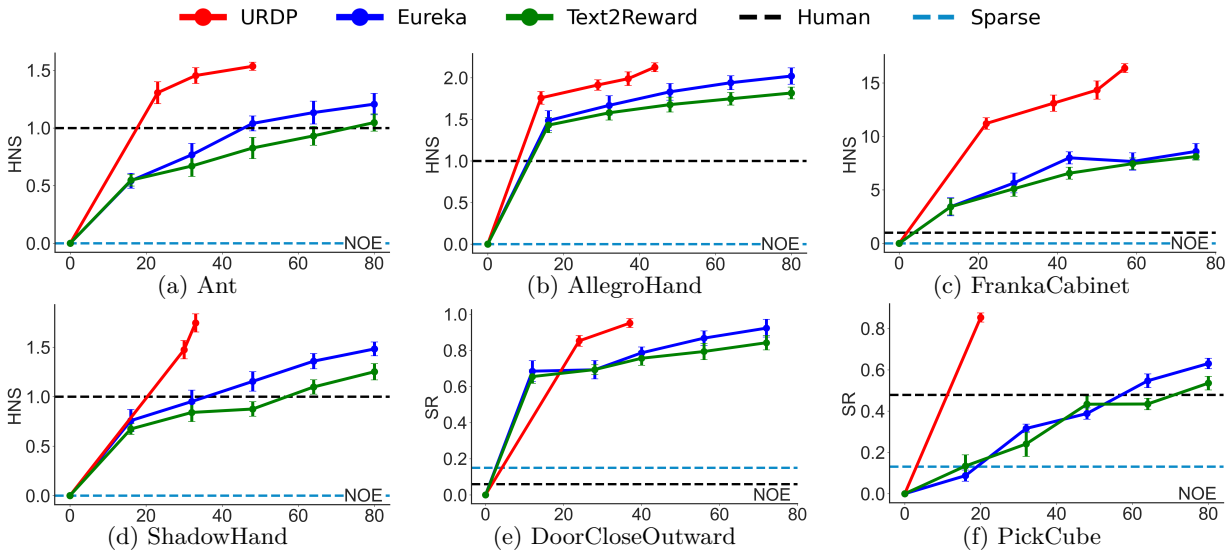


Figure 2: Comparisons of *URDP* with other methods in Isaac (a-d), Dexterity (e), and ManiSkill2 (f).

5.4 Ablation Experiments

Furthermore, we explore the role of each core content in *URDP* in achieving the above results.

Abl-1: Uncertainty quantification improves the efficiency of reward design. To evaluate the role of uncertainty sampling, we conduct ablation studies by removing the uncertainty sampling and filtering module from *URDP* (denoted as **URDP w.o. Uncertainty**). Experimental results in Figure 3 demonstrate that *URDP* achieves comparable success rates while requiring significantly fewer optimization episodes (NOE) than URDP w.o. Uncertainty, quantitatively validating the efficiency improvement brought by uncertainty-aware sampling. Interestingly, our analysis also reveals the positive role of the uncertainty in obtaining novel reward functions, with detailed mechanistic explanations to be discussed in Section 5.5 (Disc-2).

Abl-2: Decoupled optimization is the cornerstone for the collaborative improvement of performance and efficiency. This experimental investigation examines the role of decoupling in *URDP*, where reward components and their associated intensities are optimized separately. To evaluate this mechanism, we ablate UABO from *URDP* (denoted as **URDP w.o. UABO**) while maintaining identical configurations otherwise, resulting in a system where both reward components and intensities are jointly configured by the LLM without alternating optimization. Under unrestricted NLC, we compare the HNS or SR achieved by URDP w.o. UABO using equivalent NOE to the standard *URDP* implementation. As shown in Figure 4 (dashed lines), consistent reductions in both HNS and SR are observed across all three benchmark tasks, demonstrating the substantial impact of decoupled optimization on improving reward function design quality. Furthermore, *URDP* exhibites faster convergence (requiring fewer NLC) to optimal solutions, suggesting that decoupling also enhances the efficiency of evolutionary search. A comprehensive discussion of this phenomena and the underlying mechanisms is presented in Section 5.5 (Disc-1).

Abl-3: The UABO play a key role in performance improvement. This experimental study systematically compares the numerical optimization capabilities between large language models (LLMs) and Bayesian optimization approaches within our framework. In the first ablation, we replaced the UABO module with the LLM reflection (denoted as **URDP w. LLMO**), employing identical prompting strategies to Eureka,

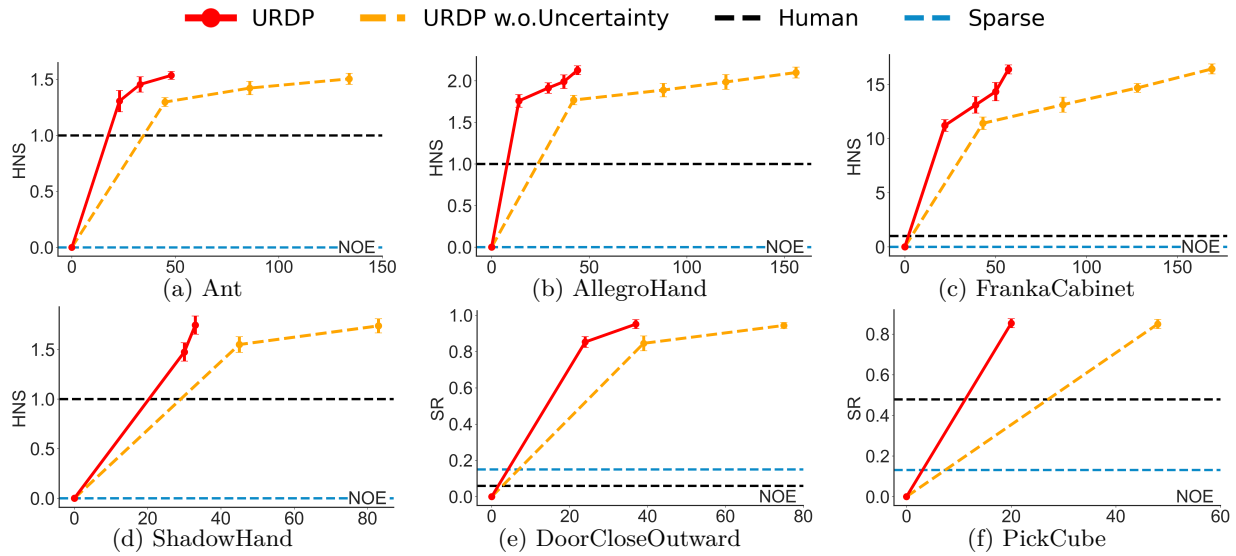


Figure 3: When generating reward functions of comparable quality, ***URDP*** requires significantly fewer simulation training episodes, attributable to its effective uncertainty-based filtering mechanism.

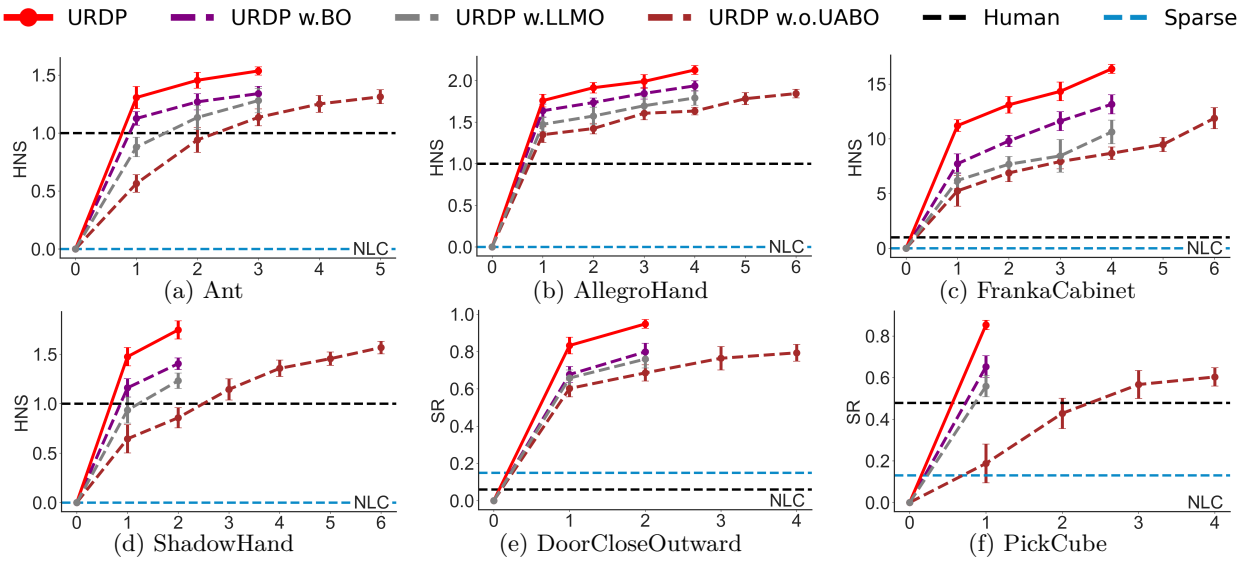


Figure 4: A comparison of the ***URDP***, URDP w. BO, URDP w. LLMO and URDP w.o. UABO when all methods utilize identical simulation budgets (NOE).

thereby configuring both outer-loop (reward components) and inner-loop (reward intensities) optimization entirely through LLMs. Figure 4 demonstrates that even under decoupled optimization conditions, LLM-based numerical optimization underperforms Bayesian optimization, revealing fundamental limitations in mathematical optimization capabilities while confirming the critical role of Bayesian methods in enhancing reward function quality. These results substantiate our core hypothesis that LLMs serve more effectively as controllers for numerical optimization tools rather than direct optimizers. Subsequent validation experiments replacing UABO with standard BO (**URDP w. BO**) reveal UABO's superior efficiency (see Figure 5): ***URDP*** (UABO) achieves comparable or better performance than URDP w. BO using only 80% of sampling budget across all Isaac tasks, with consistently superior final reward function performance, demonstrating that uncertainty-aware priors accelerate optimal search in reward function design.

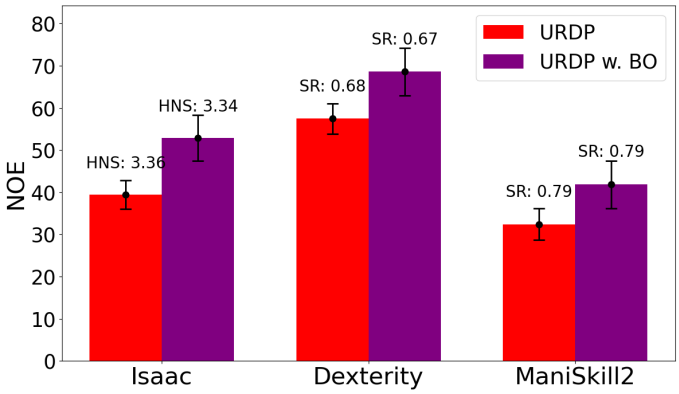
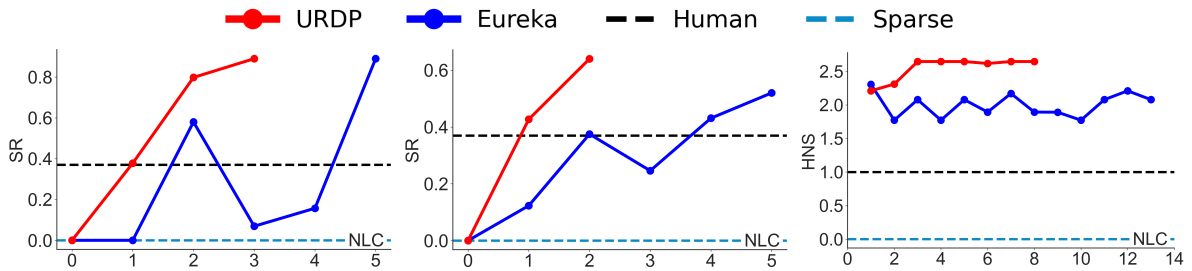


Figure 5: *URDP* achieves a significantly greater improvement in efficiency across all benchmarks compared to BO.

5.5 Extended Discussion

Disc-1: Decoupling and performance degradation in evolutionary search. As illustrated in Figure 6 (blue line), our experiments reveal performance degradation during evolutionary search in the baseline (Eureka) approach, with performance regression observed in 23% of Isaac tasks. Analysis of the LLMs' decision-making in these cases demonstrates that in 75% of instances, the models modified only the reward intensity hyperparameters while leaving the reward components unchanged, indicating a propensity for erroneous judgments in numerical optimization. More critically, we identify *oscillatory phenomenon* in the baseline optimization process (see Figure 6c), where LLMs entered persistent cycles of alternating between limited sets of hyperparameters during evolutionary search. This optimization instability resulted in complete convergence failure, with the baseline system trapped in ineffective, non-progressive iterations.

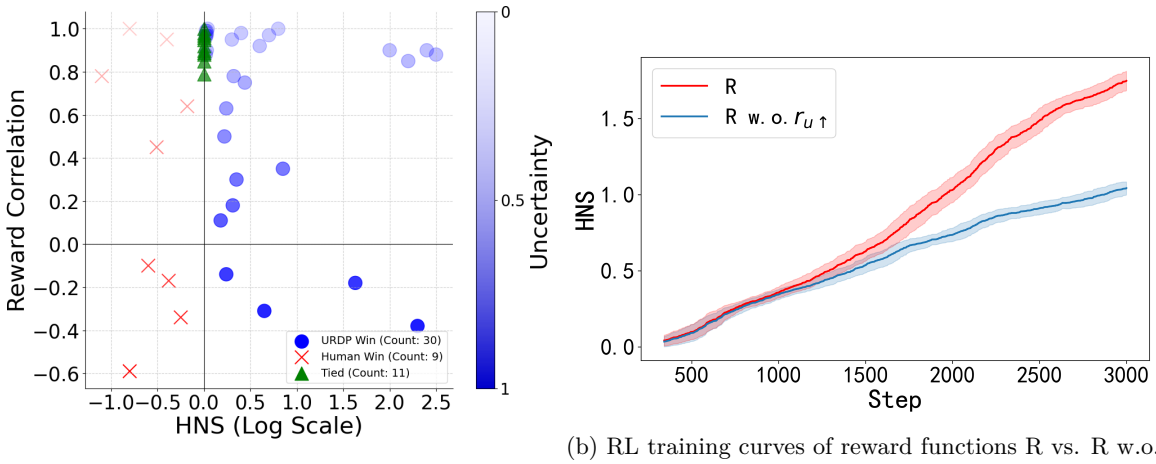


(a) LiftUnderarm (b) OpenCabinetDrawer (c) Humanoid

Figure 6: The baseline (Eureka) exhibits undesirable performance degradation during evolutionary search on certain tasks (a-b). Notably, the oscillatory phenomenon is detected in the baseline method for Task (c), indicating substantial computational waste of the baseline method.

In contrast, *URDP* demonstrates superior optimization efficiency, requiring significantly fewer evolutionary iterations (NLC) while exhibiting markedly reduced instances of performance regression and oscillatory behavior during the optimization process. Our empirical results show performance regression in merely 1% of Isaac tasks, with no observed cases of persistent optimization oscillations. These findings provide a strong empirical explanation for the effectiveness of the decoupled reward design approach implemented in *URDP*.

Disc-2: The correlation between Reward Component Uncertainty and novel reward discovery. Our investigation reveals a noteworthy phenomenon: the high-uncertainty reward components ($r_{u\uparrow}$) identified by *URDP* frequently correspond to novel reward components not previously utilized in human-designed reward functions, with these components exhibiting significant reward shaping effects during RL training, thereby revealing a dual role of the uncertainty in enhancing both optimization efficiency and final policy performance.



(a) *URDP* Rewards vs. Human Rewards on IsaacGym. $r_{u\uparrow}$ on the ShadowHand.

(b) RL training curves of reward functions R vs. R w.o.

Figure 7: (a) High-uncertainty reward components are likely novel reward components that humans have never explored before. (b) $r_{u\uparrow}$ is conducive to achieving higher returns.

Here, we refer to a reward component as $r_{u\uparrow}$ when the uncertainty of the reward is greater than 0.9. Through systematic comparisons between reward components designed by the *URDP* and conventional human-designed rewards, Figure 7 (a) illustrates a strong correlation between the component uncertainty level and the novelty, suggesting that higher uncertainty components represent more innovative reward formulations. Ablation studies conducted by removing $r_{u\uparrow}$ components from the reward function (R w.o. $r_{u\uparrow}$) and retraining PPO agents reveal statistically significant performance degradation and poorer convergence characteristics in the resulting policies as shown in Figure 7 (b), whereas the complete reward function maintains substantially better optimization stability, collectively providing conclusive evidence that $r_{u\uparrow}$ components play an essential role in effective reward shaping and policy learning regularization. See more examples in App. G.1.

6 Conclusion

In this work, we present a novel decoupled architecture for automated high-quality reward function design in reinforcement learning. Our framework introduces uncertainty quantification into reward component design, significantly improving the sampling efficiency of LLMs. Furthermore, we propose Uncertainty-Aware Bayesian Optimization to enable efficient hyperparameter search. Extensive experimental results demonstrate that our approach outperforms existing methods in both reward function quality and automated design efficiency.

References

Gustaf Ahdritz, Tian Qin, Nikhil Vyas, Boaz Barak, and Benjamin L Edelman. Distinguishing the knowable from the unknowable with language models. In *International Conference on Machine Learning (ICML)*, pp. 503–549. PMLR, 2024.

Sebastian Ament, Samuel Daulton, David Eriksson, Maximilian Balandat, and Eytan Bakshy. Unexpected improvements to expected improvement for bayesian optimization. *Advances in Neural Information Processing Systems (NeurIPS)*, 36:20577–20612, 2023.

Saurabh Arora and Prashant Doshi. A survey of inverse reinforcement learning: Challenges, methods and progress. *Artificial Intelligence*, 297:103500, 2021.

Mido Assran, Adrien Bardes, David Fan, Quentin Garrido, Russell Howes, Matthew Muckley, Ammar Rizvi, Claire Roberts, Koustuv Sinha, Artem Zholus, et al. V-jepa 2: Self-supervised video models enable understanding, prediction and planning. *arXiv preprint arXiv:2506.09985*, 2025.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Proceedings of Advances In Neural Information Processing Systems (NeurIPS)*, pp. 1877–1901, 2020.

Adam D Bull. Convergence rates of efficient global optimization algorithms. *Journal of Machine Learning Research*, 12(10), 2011.

Yuji Cao, Huan Zhao, Yuheng Cheng, Ting Shu, Yue Chen, Guolong Liu, Gaoqi Liang, Junhua Zhao, Jinyue Yan, and Yun Li. Survey on large language model-enhanced reinforcement learning: Concept, taxonomy, and methods. *IEEE Transactions on Neural Networks and Learning Systems*, 2024.

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. *Advances in Neural Information Processing Systems (NeurIPS)*, 35:5150–5163, 2022.

Fengxiang Cheng, Haoxuan Li, Fenrong Liu, Robert van Rooij, Kun Zhang, and Zhouchen Lin. Empowering llms with logical reasoning: A comprehensive survey. *arXiv preprint arXiv:2502.15652*, 2025.

Andrew Foong, David Burt, Yingzhen Li, and Richard Turner. On the expressiveness of approximate inference in bayesian neural networks. In *Proceedings of Advances In Neural Information Processing Systems (NeurIPS)*, volume 33, pp. 15897–15908, 2020.

Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2015.

Jiahui Geng, Fengyu Cai, Yuxia Wang, Heinz Koeppl, Preslav Nakov, and Iryna Gurevych. A survey of confidence estimation and calibration in large language models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 6577–6595, 2024.

Jiayuan Gu, Fanbo Xiang, Xuanlin Li, Zhan Ling, Xiqiang Liu, Tongzhou Mu, Yihe Tang, Stone Tao, Xinyue Wei, Yunchao Yao, et al. Maniskill2: A unified benchmark for generalizable manipulation skills. In *The Eleventh International Conference on Learning Representations (ICLR)*.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning (ICML)*, pp. 1861–1870. Pmlr, 2018.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Hong, Zhen Wang, Daisy Wang, and Zhiting Hu. Reasoning with language model is planning with world model. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 8154–8173, 2023.

Saurav Kadavath, Tom Conerly, Amanda Askell, Tom Henighan, Dawn Drain, Ethan Perez, Nicholas Schiefer, Zac Hatfield-Dodds, Nova DasSarma, Eli Tran-Johnson, et al. Language models (mostly) know what they know. *arXiv preprint arXiv:2207.05221*, 2022.

Lorenz Kuhn, Yarin Gal, and Sebastian Farquhar. Semantic uncertainty: Linguistic invariances for uncertainty estimation in natural language generation. In *International Conference on Learning Representations (ICLR)*, 2023.

Minae Kwon and Sang Michael. Reward design with language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Zhong-Zhi Li, Duzhen Zhang, Ming-Liang Zhang, Jiaxin Zhang, Zengyan Liu, Yuxuan Yao, Haotian Xu, Junhao Zheng, Pei-Jie Wang, Xiuyi Chen, et al. From system 1 to system 2: A survey of reasoning large language models. *arXiv preprint arXiv:2502.17419*, 2025.

Zhen Lin, Shubhendu Trivedi, and Jimeng Sun. Generating with confidence: Uncertainty quantification for black-box large language models. *Transactions on Machine Learning Research*.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

Xiaoou Liu, Tiejin Chen, Longchao Da, Chacha Chen, Zhen Lin, and Hua Wei. Uncertainty quantification and confidence calibration in large language models: A survey. *arXiv preprint arXiv:2503.15850*, 2025.

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 (ICLR)*, 2024a.

Yecheng Jason Ma, William Liang, Hungju Wang, Sam Wang, Yuke Zhu, Linxi Fan, Osbert Bastani, and Dinesh Jayaraman. Dreureka: Language model guided sim-to-real transfer. In *Robotics: Science and Systems (RSS)*, 2024b.

Christopher Mohri and Tatsunori Hashimoto. Language models with conformal factuality guarantees. In *International Conference on Machine Learning (ICML)*, pp. 36029–36047. PMLR, 2024.

F. J. Narcowich, J. D. Ward, and H. Wendland. Refined error estimates for radial basis function interpolation. *Constructive Approximation*, 2003.

Muhammad U. Nasir, Sam Earle, Christopher Cleghorn, Steven James, and Julian Togelius. Llmatic: Neural architecture search via large language models and quality diversity optimization. 2024.

OpenAI. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025. URL `https://arxiv.org/abs/2412.15115`.

Alec Radford, Rewon Child Jeffrey Wu, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 2019.

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

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

Ola Shorinwa, Zhiting Mei, Justin Lidard, Allen Z Ren, and Anirudha Majumdar. A survey on uncertainty quantification of large language models: Taxonomy, open research challenges, and future directions. *arXiv preprint arXiv:2412.05563*, 2024.

Satinder Singh, Richard L Lewis, and Andrew G Barto. Where do rewards come from. In *Proceedings of the annual conference of the cognitive science society*, pp. 2601–2606. Cognitive Science Society, 2009.

Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems (NeurIPS)*, 25, 2012.

Jiayang Song, Zhehua Zhou, Jiawei Liu, Chunrong Fang, Zhan Shu, and Lei Ma. Self-refined large language model as automated reward function designer for deep reinforcement learning in robotics. *arXiv preprint arXiv:2309.06687*, 2023.

Jiayuan Su, Jing Luo, Hongwei Wang, and Lu Cheng. Api is enough: Conformal prediction for large language models without logit-access. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 979–995, 2024.

Ziyu Wan, Xidong Feng, Muning Wen, Stephen Marcus McAleer, Ying Wen, Weinan Zhang, and Jun Wang. Alphazero-like tree-search can guide large language model decoding and training. In *Forty-first International Conference on Machine Learning (ICML)*, 2024.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Huai hsin Chi, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

Zi Wang, George E Dahl, Kevin Swersky, Chansoo Lee, Zachary Nado, Justin Gilmer, Jasper Snoek, and Zoubin Ghahramani. Pre-trained gaussian processes for bayesian optimization. *Journal of Machine Learning Research*, 25(212):1–83, 2024.

Shitao Xiao, Zheng Liu, Peitian Zhang, Niklas Muennighoff, Defu Lian, and Jian-Yun Nie. C-pack: Packed resources for general chinese embeddings. In *Proceedings of the 47th international ACM SIGIR conference on research and development in information retrieval*, pp. 641–649, 2024.

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 (ICLR)*, 2024.

Yang Yan, Yu Lu, Renjun Xu, and Zhenzhong Lan. Do phd-level llms truly grasp elementary addition? probing rule learning vs. memorization in large language models. *arXiv preprint arXiv:2504.05262*, 2025.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations (ICLR)*.

Wenhao Yu, Nimrod Gileadi, Chuyuan Fu, Sean Kirmani, Kuang-Huei Lee, Montse 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 Conference on Robot Learning (CoRL)*, 2023.

Zhuohao Yu, Weizheng Gu, Yidong Wang, Zhengran Zeng, Jindong Wang, Wei Ye, and Shikun Zhang. Outcome-refining process supervision for code generation. *arXiv preprint arXiv:2412.15118*, 2024.

Jialun Zhong, Wei Shen, Yanzeng Li, Songyang Gao, Hua Lu, Yicheng Chen, Yang Zhang, Wei Zhou, Jinjie Gu, and Lei Zou. A comprehensive survey of reward models: Taxonomy, applications, challenges, and future. *arXiv preprint arXiv:2504.12328*, 2025.

Brianna Zitkovich, Tianhe Yu, Sichun Xu, Peng Xu, Ted Xiao, Fei Xia, Jialin Wu, Paul Wohlhart, Stefan Welker, Ayzaan Wahid, et al. Rt-2: Vision-language-action models transfer web knowledge to robotic control. In *Conference on Robot Learning (CoRL)*, pp. 2165–2183. PMLR, 2023.

A Full Prompts

In this section, we provide all prompts in the *URDP* framework.

Prompt 1: Initial 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:
@torch.jit.script
def compute_reward(object_pos: torch.Tensor, goal_pos: torch.Tensor) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
    ...
    return reward, {}
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.
```

Prompt 2: Reward reflection and feedback

```
We trained a RL policy using the provided reward function code and tracked the values of the individual components in the
reward function as well as global policy metrics such as success rates and episode lengths after every {epoch_freq} epochs
and the maximum, mean, minimum values encountered:
<REWARD REFLECTION HERE1>

We calculated a score for each sample based on the uncertainty of the reward term. We then calculated the standard and
extreme deviations of all the sample scores in this iteration, which were as follows:
<REWARD REFLECTION HERE2>

 Please adopt the following recommendations for the next iteration of reward function generation:
    (1) If the standard deviation is less than 0.05 and the extreme deviation is less than 0.1, it is recommended to stop
    the iteration and instead encourage exploration of new combinations of reward terms.
    (2) For all reward items with higher scores it is recommended to keep them and for those with lower scores it is
    recommended to remove them.
    (3) Only the combination of reward items and the content of the reward function need to be optimised, not the numerical
    optimisation.

Please carefully analyze the policy feedback and provide a new, improved reward function that can better solve the task.
Some helpful tips for analyzing the policy feedback:
    (1) If the success rates are always near zero, then you must rewrite the entire reward function
    (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) Re-writing the reward component
        (c) Discarding the reward component
    (3) If some reward components' magnitude is significantly larger, then you must re-scale its value to a proper range
Please analyze each existing reward component in the suggested manner above first, and then write the reward function
code.
```

Prompt 3: Code formatting tip

```
The output of the reward function should consist of two items:
    (1) the total reward,
    (2) a dictionary of each individual reward component.
The code output should be formatted as a python code string: "```python ... ```".

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
    overall reward or its components
    (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
    (3) Make sure the type of each input variable is correctly specified; a float input variable should not be specified
    as torch.Tensor
    (4) Most importantly, the reward code's input variables must contain only attributes of the provided environment class
    definition (namely, variables that have prefix self.). Under no circumstance can you introduce new input variables.
```

B Benchmark Details

B.1 An Introduction to the Benchmarks

Isaac.The Isaac Gym benchmark includes a broad set of continuous control tasks, covering locomotion, balancing, aerial control, and dexterous manipulation. Robots range from low-DoF systems (e.g., Cartpole, Ball Balance) to complex agents such as Humanoid, Anymal, AllegroHand, and ShadowHand. Each task presents different control challenges, requiring precise joint coordination, stable gait generation, or fine-grained object interaction. All tasks provide observations including joint positions, velocities, root orientation, and task-specific data such as object pose or goal location. The control mode varies by task—torque, velocity, or end-effector control—depending on the robot type. Randomization of initial states and physical parameters (e.g., mass, friction) is applied during training to improve robustness and generalization. The benchmark emphasizes both low-level motor control and high-level strategy in physics-rich environments.

Dexterity.The Dexterous benchmark focuses on dexterous manipulation using the 24-DoF ShadowHand across a wide range of object-centric tasks. These include stacking blocks, turning faucets, opening doors, rotating bottles, catching objects, and tool use. The tasks require precise finger control, contact-rich interactions, and adaptability to diverse object geometries and behaviors. Each environment provides proprioceptive input (joint states, fingertip positions), as well as object-related observations (pose, velocity, goal state). Control is applied via joint position or velocity commands. The tasks involve significant variability in object placement, orientation, and physical properties, encouraging the development of general and robust manipulation policies. The benchmark highlights the challenge of high-dimensional motor coordination in real-world-like, unstructured settings.

Maniskill2. In the ManiSkill2 environment, a 7-DoF Franka Panda robotic arm is used by default. For tasks focused on stationary manipulation—such as Lift Cube, Pick Cube, Turn Faucet, and Stack Cube—a fixed-base arm configuration is employed. In contrast, tasks involving mobility, such as Open Cabinet Door and Open Cabinet Drawer, utilize a single-arm robot mounted on a Sciurus17 mobile base. The Push Chair task is handled by a dual-arm system, also equipped with the Sciurus17 base.The observation space includes robot-centric data like joint angles, joint velocities, and the base's pose (position and orientation in the world frame), along with task-specific inputs such as goal coordinates and end-effector locations. Control is performed in end-effector delta pose mode, which directly manages changes in 3D translation and orientation, the latter expressed in axis-angle form relative to the end-effector's frame.Each task features variability in key parameters, including the initial and goal states of the object being manipulated, the robot's starting joint configuration, and physical dynamics like friction and damping.

B.2 ManiSkill2 Environment Details

We provide detailed information about the ManiSkill2 environment in this section. Detailed information about the Isaac and Dexterity environments is the same as in the Eureka (see the content in the appendix of the paper Ma et al. (2024a)). For each environment, we list its observation and action dimensions, the original description of the task, and the task fitness function F.

ManiSkill2 Environments
Environment (obs dim, action dim) Task description Task fitness function F
`PickCube-v0` (51,7) This class corresponds to the PickCube task in ManiSkill. This environment consists of a robot arm and a cube placed on the table. At the beginning, the cube appears at a random location and orientation. The agent must control the gripper to approach, grasp, and lift the cube above a threshold height. The challenge lies in object localization, precise control, and stable grasping. 1[dist_cube_goal < 0.05]
`LiftCube-v0` (42,7) This environment corresponds to the LiftCube task. The agent is required to grasp a cube and lift it vertically above a specific height threshold. The task emphasizes accurate vertical movement and stable grasping without disturbing the cube's pose. 1[cube_height > 0.2]
`TurnFaucet-v0` (40, 7) This class corresponds to the TurnFaucet task. A faucet handle is mounted on a wall, and the agent must rotate it clockwise or counterclockwise to a target angle. The challenge lies in establishing proper contact, applying sufficient torque, and maintaining stability during the turning motion. 1[rotation_reward < 0.1]
`OpenCabinetDoor-v1` (75, 11) This environment corresponds to the OpenCabinetDoor task. A cabinet with a side-hinged door is presented. The agent must locate and pull the door handle to open it. The task involves estimating the door's hinge axis, approaching from an appropriate angle, and applying a pulling force that aligns with the door's rotation. 1[goal_diff < 0.1 and is_static]
`OpenCabinetDrawer-v1` (75, 11) This class corresponds to the OpenCabinetDrawer task. The robot must open a drawer embedded in a cabinet by locating the handle and pulling it outward. The task requires both accurate handle grasping and force application along a linear trajectory, while avoiding excessive torque that could misalign the drawer. 1[goal_diff < 0.05 and is_static]
`PushChair-v1` (131, 18) This environment corresponds to the PushChair task. The robot must push a movable chair from its initial location to a designated target region. The chair is free to rotate and slide. The agent needs to make strategic contact with the chair body and adjust its pushing direction dynamically to avoid misalignment and ensure accurate placement. 1[chair_to_target_dist < 0.3 and chair_tilt < 0.2]

C Implementation Details

C.1 Implementation Details of Sampling and Uncertainty

When explicitly prompted to generate diverse outputs, LLMs inevitably produce varying textual expressions for semantically equivalent content - a phenomenon particularly evident in reward function generation where code implementations may differ lexically while encoding identical reward semantics. For instance, as demonstrated in Section D.2, two LLM-generated reward function samples might both incorporate velocity-based rewards while exhibiting completely different textual formulations. Failure to detect and eliminate such semantic redundancies leads to computationally expensive duplicate evaluations that cannot be effectively identified through surface-level text matching, necessitating deeper semantic analysis for accurate deduplication.

Therefore, the *URDP* utilizes the BGE-M3 model Xiao et al. (2024) for the purpose of semantic similarity assessment, whereas the built-in SequenceMatcher in Python is employed for text similarity assessment. The uncertainty quantification for both reward components and reward functions is implemented through similarity comparison. Specifically, the component uncertainty score ($U(r_{i,:})$) is computed by comparing a given reward component against all components generated within the same iteration. The reward function uncertainty ($U(R_i)$) score is derived through comparison with all functionally similar reward functions from the same iteration (referred to as a similarity group). From each similarity group, only one reward function is randomly selected for training, while the remaining ones are discarded, thereby filtering out redundant reward functions. See Alg. 2 for the pseudocode. We employ a similarity threshold of 0.95, where the final similarity metric is determined as the maximum value between semantic similarity and textual similarity.

Algorithm 2: Uncertainty Quantification in the *URDP*

Input: K reward component samples $\{r_{i,1}, r_{i,2}, \ldots, r_{i,m}\}_{i \in K}$, text models S_{text} and semantic models $S_{semantic}$.

Output: Reward components uncertainty $\{U(r_{i,1}), \ldots, U(r_{i,m})\}$, reward functions uncertainty $U(R_i)$ and similarity sample group B_i.

foreach $R_{i \in K}$ **do**
  foreach $r_{i,1}, r_{i,2}, \ldots, r_{i,m}$ **do**
    foreach $r_{j,1}, r_{j,2}, \ldots, r_{j \in k,m}, j > i$ **do**
      if *max*($S_{text}(r_{i,m}, r_{j,m})$, $S_{semantic}(r_{i,m}, r_{j,m})$)>0.95 **then**
        count$_m$+1
    $U(r_{i,m})$=1-count$_m$/K
  $U(R_i) = U(r_i)/(U(r_i) + \cdots + U(r_k))$
  foreach $R_{j>i}$ **do**
    if *max*($S_{text}(R_i, R_j)$, $S_{semantic}(R_i, R_j)$)>0.95 *and* R_i *not in other* B **then**
      Add R_j to B_i

return $\{U(r_{i,1}), \ldots, U(r_{i,m})\}$, $U(R_i)$, $B_{i,n=1}$.

C.2 Hyper-parameter Settings

All hyperparameters in *URDP* are listed in Table 3. The reinforcement learning algorithms employed for validation maintain the default configurations specified for each respective environment, with all hyperparameters comprehensively documented in Tables 4 and 5.

Table 3: Hyperparameters of *URDP*.

Hyper-parameter	**Value**
Quantity of the reward samples K	16
Maximum # of iterations N_{outer}	10
Baseline # of iterations N_{inner}	10
maximum similarity ω	0.95

D Case Studies

D.1 Case Study 1: LLMs in Numerical Optimization

This study employs two comparative examples to visualize the sdifferences between *URDP* and Eureka in optimization processes. Example 1(a) and 1(b) present the respective design trajectories of *URDP* and Eureka for the ShadowHand task, where red annotations denote reward components and blue text indicates reward intensity hyperparameters.

Table 4: Hyperparameters of SAC algorithm applied to Maniskill2.

Hyper-parameter	**Value**
Discount factor γ	0.95
Target update frequency	1
Learning rate	3×10^{-4}
Train frequency	8
Soft update τ	5×10^{-3}
Gradient steps	4
Learning starts	4000
Hidden units per layer	256
Batch Size	1024
# of layers	2
Initial temperature	0.2
Rollout steps per episode	200

Table 5: Hyperparameters of PPO algorithm applied to each task.

Hyper-parameter	**Value**
Discount factor γ	0.99 (Isaac), 0.96 (Dexterity), 0.85 (ManiSkill2)
# of epochs per update	8 (Isaac), 5 (Dexterity), 15 (ManiSkill2)
Learning rate	5×10^{-4} (Isaac), 3×10^{-4} (Dexterity, ManiSkill2)
Batch size	32768, 16384, 8192 (Isaac), 16384 (Dexterity), 400 (ManiSkill2)
Target KL divergence	0.008 (Isaac), 0.016 (Dexterity), 0.05 (ManiSkill2)
# of layers	3 (Isaac, Dexterity), 2 (ManiSkill2)
# of steps per update	16 (Isaac), 8 (Dexterity), 3200 (ManiSkill2)

Analysis of Example 1(a) demonstrates that during evolutionary search iterations, Eureka exclusively modifies reward intensity hyperparameters while failing to improve reward components. Despite multiple optimization attempts, this approach yields degraded performance. This finding reveals a critical limitation: when simultaneously optimizing both reward components and their strengths, LLMs cannot effectively utilize their inherent advantages in semantic correlation and autoregressive modeling, while their deficiencies in numerical optimization become particularly pronounced.

In contrast, ***URDP***'s decoupled alternating optimization demonstrates systematic effectiveness. As shown in Example 1(b), ***URDP*** proactively explores novel reward components (e.g., replacing "distance reward" with "velocity reward"), achieving a 70.3% performance improvement (from 0.929 to 1.583) before hyperparameter optimization. Subsequent Bayesian optimization further enhances performance with an average 22.1% gain, culminating in a 102.5% total improvement over the initial configuration. Example 2 confirms this consistent advantage, providing compelling visual evidence of ***URDP***'s superior optimization characteristics.

Example 1(a): Eureka reward functions on ShadowHand (Iteration 2, HNS: 1.045; Iteration 3, HNS: 0.898; Iteration 4, HNS: 1.019; Iteration 5 HNS: 0.955).

```
Iteration 2:
def compute_reward(object_rot: torch.Tensor, goal_rot: torch.Tensor, object_angvel: torch.Tensor) -> Tuple[torch.Tensor,
Dict[str, torch.Tensor]]:
    # Temperature parameters for reward components
    orientation_temp: float = 5.0  # Increased sensitivity for orientation alignment
    angular_velocity_temp: float = 0.5  # Adjusted sensitivity for angular velocity

    # Compute the difference in orientation between the object and the goal
    orientation_diff = 1.0 - torch.abs(torch.sum(object_rot * goal_rot, dim=-1))  # Quaternion dot product

    # Reward for minimizing the orientation difference
    orientation_reward = torch.exp(-orientation_temp * orientation_diff)
```

```
    # Reward for spinning the object (encouraging non-zero angular velocity)
    angular_velocity_magnitude = torch.norm(object_angvel, dim=-1)
    angular_velocity_reward = torch.tanh(angular_velocity_temp * angular_velocity_magnitude)  # Rescaled to [-1, 1]

    # Combine the rewards with a focus on orientation alignment
    total_reward = orientation_reward * (1.0 + angular_velocity_reward)  # Weighted combination

    # Return the total reward and individual components
    reward_components = {
        "orientation_reward": orientation_reward,
        "angular_velocity_reward": angular_velocity_reward
    }
    return total_reward, reward_components

#########################################################################################################

Iteration 3:
def compute_reward(object_rot: torch.Tensor, goal_rot: torch.Tensor, object_angvel: torch.Tensor) -> Tuple[torch.Tensor,
Dict[str, torch.Tensor]]:
    # Temperature parameters for reward components
    orientation_temp: float = 10.0  # Increased sensitivity for orientation alignment
    angular_velocity_temp: float = 1.0  # Adjusted sensitivity for angular velocity

    # Compute the difference in orientation between the object and the goal
    orientation_diff = 1.0 - torch.abs(torch.sum(object_rot * goal_rot, dim=-1))  # Quaternion dot product

    # Reward for minimizing the orientation difference
    orientation_reward = torch.exp(-orientation_temp * orientation_diff)

    # Reward for spinning the object (encouraging non-zero angular velocity)
    angular_velocity_magnitude = torch.norm(object_angvel, dim=-1)
    angular_velocity_reward = torch.tanh(angular_velocity_temp * angular_velocity_magnitude)  # Rescaled to [-1, 1]

    # Combine the rewards with a focus on orientation alignment
    total_reward = orientation_reward * (1.0 + angular_velocity_reward)  # Weighted combination

    # Return the total reward and individual components
    reward_components = {
        "orientation_reward": orientation_reward,
        "angular_velocity_reward": angular_velocity_reward
    }
    return total_reward, reward_components

#########################################################################################################

Iteration 4:
def compute_reward(object_rot: torch.Tensor, goal_rot: torch.Tensor, object_angvel: torch.Tensor) -> Tuple[torch.Tensor,
Dict[str, torch.Tensor]]:
    # Temperature parameters for reward components
    orientation_temp: float = 10.0  # Increased sensitivity for orientation alignment
    angular_velocity_temp: float = 2.0  # Adjusted sensitivity for angular velocity

    # Compute the difference in orientation between the object and the goal
    orientation_diff = 1.0 - torch.abs(torch.sum(object_rot * goal_rot, dim=-1))  # Quaternion dot product

    # Reward for minimizing the orientation difference
    orientation_reward = torch.exp(-orientation_temp * orientation_diff)

    # Reward for spinning the object (encouraging non-zero angular velocity)
    angular_velocity_magnitude = torch.norm(object_angvel, dim=-1)
    angular_velocity_reward = torch.tanh(angular_velocity_temp * angular_velocity_magnitude)  # Rescaled to [-1, 1]

    # Combine the rewards with a focus on orientation alignment
    total_reward = orientation_reward * (1.0 + angular_velocity_reward)  # Weighted combination

    # Return the total reward and individual components
    reward_components = {
        "orientation_reward": orientation_reward,
        "angular_velocity_reward": angular_velocity_reward
    }
    return total_reward, reward_components

#########################################################################################################

Iteration 5:
def compute_reward(object_rot: torch.Tensor, goal_rot: torch.Tensor, object_angvel: torch.Tensor) -> Tuple[torch.Tensor,
Dict[str, torch.Tensor]]:
    # Temperature parameters for reward components
    orientation_temp: float = 50.0  # Increased sensitivity for orientation alignment
```

```
    angular_velocity_temp: float = 2.0  # Adjusted sensitivity for angular velocity

    # Compute the difference in orientation between the object and the goal
    orientation_diff = 1.0 - torch.abs(torch.sum(object_rot * goal_rot, dim=-1))  # Quaternion dot product

    # Reward for minimizing the orientation difference
    orientation_reward = torch.exp(-orientation_temp * orientation_diff)

    # Reward for spinning the object (encouraging non-zero angular velocity)
    angular_velocity_magnitude = torch.norm(object_angvel, dim=-1)
    angular_velocity_reward = torch.tanh(angular_velocity_temp * angular_velocity_magnitude)  # Rescaled to [-1, 1]

    # Combine the rewards with a focus on orientation alignment
    total_reward = orientation_reward * (1.0 + angular_velocity_reward)  # Weighted combination

    # Return the total reward and individual components
    reward_components = {
        "orientation_reward": orientation_reward,
        "angular_velocity_reward": angular_velocity_reward
    }
    return total_reward, reward_components
```

Example 1(b): *URDP* reward functions on ShadowHand. Iteration 1 (before UABO), HNS: 0.929; Iteration 1 (after UABO), HNS: 1.165; Iteration 2 (before UABO), HNS: 1.583; Iteration 2 (after UABO), HNS: 1.882.

```
Iteration 1(before UABO):
def compute_reward(object_rot: torch.Tensor, goal_rot: torch.Tensor) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
    # Temperature parameters
    orientation_temp = 1.0  # Reduced temperature for better sensitivity
    distance_temp = 1.0     # Temperature for distance-based reward

    # Compute the dot product between the object and goal quaternions
    dot_product = torch.sum(object_rot * goal_rot, dim=1)

    # Ensure the dot product is within the valid range [-1, 1]
    dot_product = torch.clamp(dot_product, -1.0, 1.0)

    # Compute the angle difference between the quaternions
    angle_diff = torch.acos(2.0 * dot_product**2 - 1.0)

    # Orientation reward: exponential transformation of the angle difference
    orientation_reward = torch.exp(-orientation_temp * angle_diff)

    # Distance-based reward: encourages reducing the angle difference
    distance_reward = -angle_diff  # Negative because we want to minimize the difference

    # Success bonus: reward for achieving the target orientation
    success_threshold: float = 0.05  # Easier threshold for success
    success_bonus = torch.where(angle_diff < success_threshold, 100.0, 0.0)  # Larger bonus

    # Total reward: weighted sum of orientation reward, distance reward, and success bonus
    total_reward = orientation_reward + distance_reward + success_bonus

    # Dictionary of individual reward components
    reward_components = {
        "orientation_reward": orientation_reward,
        "distance_reward": distance_reward,
    }

    return total_reward, reward_components

#########################################################################################################

Iteration 1(after UABO):
def compute_reward(object_rot: torch.Tensor, goal_rot: torch.Tensor) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
    # Temperature parameters
    orientation_temp = 1.1134  # Reduced temperature for better sensitivity
    distance_temp = 1.1134     # Temperature for distance-based reward

    # Compute the dot product between the object and goal quaternions
    dot_product = torch.sum(object_rot * goal_rot, dim=1)

    # Ensure the dot product is within the valid range [-1, 1]
    dot_product = torch.clamp(dot_product, -1.0, 1.0)

    # Compute the angle difference between the quaternions
```

```
    angle_diff = torch.acos(2.0 * dot_product**2 - 1.0)

    # Orientation reward: exponential transformation of the angle difference
    orientation_reward = torch.exp(-orientation_temp * angle_diff)

    # Distance-based reward: encourages reducing the angle difference
    distance_reward = -angle_diff  # Negative because we want to minimize the difference

    # Success bonus: reward for achieving the target orientation
    success_threshold: float = 0.05  # Easier threshold for success
    success_bonus = torch.where(angle_diff < success_threshold, 100.0, 0.0)  # Larger bonus

    # Total reward: weighted sum of orientation reward, distance reward, and success bonus
    total_reward = orientation_reward + distance_reward + success_bonus

    # Dictionary of individual reward components
    reward_components = {
        "orientation_reward": orientation_reward,
        "distance_reward": distance_reward
    }

    return total_reward, reward_components

#########################################################################################################

Iteration 2(before UABO):
def compute_reward(object_rot: torch.Tensor, goal_rot: torch.Tensor, object_angvel: torch.Tensor) -> Tuple[torch.Tensor,
Dict[str, torch.Tensor]]:
    # Temperature parameters for reward components
    orientation_temp = 1
    velocity_temp = 0.1

    # Compute the difference in orientation between the object and the goal
    orientation_diff = torch.norm(object_rot - goal_rot, dim=-1)

    # Compute the angular velocity magnitude of the object
    angvel_magnitude = torch.norm(object_angvel, dim=-1)

    # Reward for minimizing the orientation difference
    orientation_reward = torch.exp(-orientation_temp * orientation_diff)

    # Reward for maintaining a high angular velocity (encourages spinning)
    velocity_reward = torch.exp(-velocity_temp * (1.0 / (angvel_magnitude + 1e-6)))

    # Combine the rewards with appropriate weights
    total_reward = 0.7 * orientation_reward + 0.3 * velocity_reward

    # Dictionary of individual reward components for logging
    reward_dict = {
        "orientation_reward": orientation_reward,
        "velocity_reward": velocity_reward
    }

    return total_reward, reward_dict

#########################################################################################################

Iteration 2(after UABO):
def compute_reward(object_rot: torch.Tensor, goal_rot: torch.Tensor, object_angvel: torch.Tensor) -> Tuple[torch.Tensor,
Dict[str, torch.Tensor]]:
    # Temperature parameters for reward components
    orientation_temp = 1.2642
    velocity_temp = 0.3145

    # Compute the difference in orientation between the object and the goal
    orientation_diff = torch.norm(object_rot - goal_rot, dim=-1)

    # Compute the angular velocity magnitude of the object
    angvel_magnitude = torch.norm(object_angvel, dim=-1)

    # Reward for minimizing the orientation difference
    orientation_reward = torch.exp(-orientation_temp * orientation_diff)

    # Reward for maintaining a high angular velocity (encourages spinning)
    velocity_reward = torch.exp(-velocity_temp * (1.0 / (angvel_magnitude + 1e-6)))

    # Combine the rewards with appropriate weights
    total_reward = 0.7 * orientation_reward + 0.3 * velocity_reward

    # Dictionary of individual reward components for logging
```

```
    reward_dict = {
        "orientation_reward": orientation_reward,
        "velocity_reward": velocity_reward
    }

    return total_reward, reward_dict
```

Example 2 (a): Eureka reward functions on Humanoid (Iteration 1, HNS: 2.273; Iteration 2, score: 0.412; Iteration 3, HNS: 0.032; Iteration 4, HNS: 0.127).

```
Iteration 1:
def compute_reward(root_states: torch.Tensor, dt: float) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
    # Extract the velocity of the humanoid's torso (root_states[:, 7:10] contains the linear velocity)
    velocity = root_states[:, 7:10]

    # Compute the forward speed (we assume the humanoid is moving along the x-axis)
    forward_speed = velocity[:, 0]

    # Reward for forward speed (scaled to a smaller range)
    speed_temp = 1.0  # Reduced temperature for better scaling
    speed_reward = forward_speed * speed_temp

    # Reward for consistency (encourage maintaining high speed)
    consistency_temp = 1.0
    consistency_reward = torch.exp(-consistency_temp * torch.abs(forward_speed - torch.mean(forward_speed)))

    # Total reward combines speed reward and consistency reward
    reward = speed_reward + consistency_reward

    # Dictionary of individual reward components
    reward_dict = {
        "speed_reward": speed_reward,
        "consistency_reward": consistency_reward,
    }

    return reward, reward_dict

##########################################################################################################

Iteration 2:
def compute_reward(root_states: torch.Tensor, dt: float) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
    # Extract the velocity of the humanoid's torso (root_states[:, 7:10] contains the linear velocity)
    velocity = root_states[:, 7:10]

    # Compute the forward speed (we assume the humanoid is moving along the x-axis)
    forward_speed = velocity[:, 0]

    # Reward for forward speed (scaled to a smaller range)
    speed_temp = 0.2  # Further reduced temperature for better scaling
    speed_reward = forward_speed * speed_temp

    # Reward for consistency (encourage maintaining high speed over time)
    consistency_temp = 0.5  # Increased temperature for better scaling
    consistency_reward = torch.exp(-consistency_temp * torch.abs(forward_speed - torch.mean(forward_speed)))

    # Total reward combines speed reward and consistency reward
    reward = speed_reward + consistency_reward

    # Dictionary of individual reward components
    reward_dict = {
        "speed_reward": speed_reward,
        "consistency_reward": consistency_reward,
    }

    return reward, reward_dict

##########################################################################################################

Iteration 3:
def compute_reward(root_states: torch.Tensor, dt: float) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
    # Extract the velocity of the humanoid's torso (root_states[:, 7:10] contains the linear velocity)
    velocity = root_states[:, 7:10]

    # Compute the forward speed (we assume the humanoid is moving along the x-axis)
    forward_speed = velocity[:, 0]

```

```
    # Reward for forward speed (scaled to a smaller range)
    speed_temp = 0.1  # Reduced temperature for better scaling
    speed_reward = forward_speed * speed_temp

    # Reward for consistency (encourage maintaining high speed)
    consistency_temp = 0.5  # Increased temperature for better scaling
    consistency_reward = torch.exp(-consistency_temp * torch.abs(forward_speed - torch.mean(forward_speed)))

    # Total reward combines speed reward and consistency reward
    reward = speed_reward + consistency_reward

    # Dictionary of individual reward components
    reward_dict = {
        "speed_reward": speed_reward,
        "consistency_reward": consistency_reward,
    }

    return reward, reward_dict

#########################################################################################################

Iteration 4:
def compute_reward(root_states: torch.Tensor, dt: float) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
    # Extract the velocity of the humanoid's torso (root_states[:, 7:10] contains the linear velocity)
    velocity = root_states[:, 7:10]

    # Compute the forward speed (we assume the humanoid is moving along the x-axis)
    forward_speed = velocity[:, 0]

    # Reward for forward speed (scaled to a smaller range)
    speed_temp = 0.1  # Further reduced temperature for better scaling
    speed_reward = forward_speed * speed_temp

    # Reward for consistency (encourage maintaining high speed over time)
    consistency_temp = 10  # Increased temperature for better scaling
    consistency_reward = torch.exp(-consistency_temp * torch.abs(forward_speed - torch.mean(forward_speed)))

    # Total reward combines speed reward and consistency reward
    reward = speed_reward + consistency_reward

    # Dictionary of individual reward components
    reward_dict = {
        "speed_reward": speed_reward,
        "consistency_reward": consistency_reward,
    }

    return reward, reward_dict
```

Example 2 (b): *URDP* reward functions on Humanoid. Iteration 1 (before UABO), HNS: 1.428, Iteration 1 (after UABO), HNS: 2.132; Iteration 2 (before UABO), HNS: 2.481, Iteration 2 (after UABO), HNS: 2.646.

```
Iteration 1(before UABO):
def compute_reward(root_states: torch.Tensor, dt: float) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
    # Extract the velocity of the humanoid's torso from the root states
    velocity = root_states[:, 7:10]

    # Compute the forward speed (assuming x-axis is the forward direction)
    forward_speed = velocity[:, 0]

    # Scale up the speed reward to provide stronger feedback
    speed_reward_temp = 10  # Temperature parameter for speed reward
    speed_reward = torch.exp(speed_reward_temp * forward_speed * dt)

    # Optional: Add a reward for sustained running
    sustained_speed_temp = 0.1  # Temperature parameter for sustained speed reward
    sustained_speed_reward = torch.exp(sustained_speed_temp * forward_speed * dt)

    # Combine the rewards
    total_reward = speed_reward + sustained_speed_reward

    # Return the total reward and the individual reward components
    reward_dict = {
        "speed_reward": speed_reward,
        "sustained_speed_reward": sustained_speed_reward
    }

```

```
    return total_reward, reward_dict

#########################################################################################################

Iteration 1(after UABO):
def compute_reward(root_states: torch.Tensor, dt: float) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
    # Extract the velocity of the humanoid's torso from the root states
    velocity = root_states[:, 7:10]

    # Compute the forward speed (assuming x-axis is the forward direction)
    forward_speed = velocity[:, 0]

    # Scale up the speed reward to provide stronger feedback
    speed_reward_temp = 10.4424  # Temperature parameter for speed reward
    speed_reward = torch.exp(speed_reward_temp * forward_speed * dt)

    # Optional: Add a reward for sustained running
    sustained_speed_temp = 0.3606  # Temperature parameter for sustained speed reward
    sustained_speed_reward = torch.exp(sustained_speed_temp * forward_speed * dt)

    # Combine the rewards
    total_reward = speed_reward + sustained_speed_reward

    # Return the total reward and the individual reward components
    reward_dict = {
        "speed_reward": speed_reward,
        "sustained_speed_reward": sustained_speed_reward
    }

    return total_reward, reward_dict

#########################################################################################################

Iteration 2(before UABO):
def compute_reward(root_states: torch.Tensor, dt: float) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
    # Extract the velocity of the humanoid's torso from the root states
    velocity = root_states[:, 7:10]
    torso_height = root_states[:, 2]  # Torso height (z-axis)

    # Compute the forward speed (assuming x-axis is the forward direction)
    forward_speed = velocity[:, 0]

    # Apply exponential transformation to the speed reward with a temperature parameter
    speed_temp = 1  # Temperature parameter for speed reward
    speed_reward = torch.exp(speed_temp * forward_speed) * dt

    # Penalty for falling (torso height below a threshold)
    fall_threshold = 0.8  # Minimum torso height to avoid falling
    fall_penalty = -10.0 * (torso_height < fall_threshold).float() * dt

    # Combine the rewards
    total_reward = speed_reward + fall_penalty

    # Return the total reward and the individual reward components
    reward_dict = {
        "speed_reward": speed_reward
    }

    return total_reward, reward_dict

#########################################################################################################

Iteration 2(after UABO):
def compute_reward(root_states: torch.Tensor, dt: float) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
    # Extract the velocity of the humanoid's torso from the root states
    velocity = root_states[:, 7:10]
    torso_height = root_states[:, 2]  # Torso height (z-axis)

    # Compute the forward speed (assuming x-axis is the forward direction)
    forward_speed = velocity[:, 0]

    # Apply exponential transformation to the speed reward with a temperature parameter
    speed_temp = 0.7111  # Temperature parameter for speed reward
    speed_reward = torch.exp(speed_temp * forward_speed) * dt

    # Penalty for falling (torso height below a threshold)
    fall_threshold = 0.8  # Minimum torso height to avoid falling
    fall_penalty = -10.0 * (torso_height < fall_threshold).float() * dt

    # Combine the rewards
```

```
    total_reward = speed_reward + fall_penalty

    # Return the total reward and the individual reward components
    reward_dict = {
        "speed_reward": speed_reward
    }

    return total_reward, reward_dict
```

D.2 Case Study 2: the Redundancy in Reward Function Samples

This case study analyzes two representative examples from Eureka within a single iteration, visually demonstrating a major source of computational redundancy. Our analysis reveals that Eureka generates multiple semantically equivalent but syntactically varied reward functions within a single iteration, all sharing identical reward intensities.

For instance, in Example 3, although the two reward functions (Sample #9 and #15) from iteration 1 exhibit different textual expressions ("*forward_velocity_reward*" vs. "*velocity_reward*"), their underlying reward objectives and logic are fundamentally identical. This observation suggests that the apparent diversity among Eureka-generated samples may be primarily lexical rather than semantic. Effective filtering of such pseudo-diversity is therefore essential to eliminate redundant and computationally inefficient evaluations. Example 4 is a similar example.

Example 3: The reward function codes of the task Ant after Iteration 1. The scores of Sample #9 and Sample #15 are both 2.012.

```
Iteration 1 (sample #9):
def compute_reward(root_states: torch.Tensor, actions: torch.Tensor) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:

    # Define weight parameters
    forward_velocity_temp: float = 1.0
    action_penalty_temp: float = 0.01

    # Extract forward velocity (x-axis velocity in the world frame)
    forward_velocity = root_states[:, 7]  # Velocity along the x-axis

    # Reward for forward velocity
    forward_velocity_reward = forward_velocity * forward_velocity_temp

    # Penalty for large actions to encourage energy efficiency
    action_penalty = -torch.sum(torch.square(actions), dim=-1) * action_penalty_temp

    # Total reward
    reward = forward_velocity_reward + action_penalty

    # Individual reward components
    reward_dict = {
        "forward_velocity_reward": forward_velocity_reward,
        "action_penalty": action_penalty,
    }

    return reward, reward_dict

#########################################################################################################

Iteration 1 (sample #15):
def compute_reward(root_states: torch.Tensor, actions: torch.Tensor) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:

    # Define weight parameters
    velocity_temp: float = 1.0
    action_penalty_temp: float = 0.01

    # Extract the forward velocity (x-axis velocity)
    forward_velocity = root_states[:, 7]  # x-axis velocity is at index 7

    # Reward for moving forward fast
    velocity_reward = forward_velocity * velocity_temp

    # Penalty for large actions to encourage energy efficiency
    action_penalty = -torch.sum(torch.square(actions), dim=-1) * action_penalty_temp
```

```
    # Total reward
    total_reward = velocity_reward + action_penalty

    # Individual reward components for debugging and analysis
    reward_components = {
        "velocity_reward": velocity_reward,
        "action_penalty": action_penalty,
    }

    return total_reward, reward_components
```

Example 4: The reward function codes of the task Ant after Iteration 1. The scores of Sample #3 and Sample #14 are both 0.059.

```
Iteration 1 (sample #3):
def compute_reward(object_rot: torch.Tensor, goal_rot: torch.Tensor, object_angvel: torch.Tensor) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
    # Temperature parameters for reward components
    orientation_temp: float = 1.0
    angular_vel_temp: float = 0.1

    # Compute the difference in orientation using quaternion distance
    quat_diff = quat_mul(object_rot, quat_conjugate(goal_rot))
    orientation_error = torch.norm(quat_diff[:, 1:4], dim=1)  # Ignore the scalar part for distance
    orientation_reward = torch.exp(-orientation_temp * orientation_error)

    # Penalize excessive angular velocity
    angular_vel_penalty = torch.exp(-angular_vel_temp * angular_vel_magnitude)

    # Combine rewards
    total_reward = orientation_reward * angular_vel_penalty

    # Return the total reward and individual components for debugging
    reward_components = {
        "orientation_reward": orientation_reward,
        "angular_vel_penalty": angular_vel_penalty
    }
    return total_reward, reward_components

##########################################################################################################

Iteration 1 (sample #14):
def compute_reward(object_rot: torch.Tensor, goal_rot: torch.Tensor, object_angvel: torch.Tensor) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
    # Temperature parameters for reward components
    orientation_temp = 1.0
    angular_velocity_temp = 0.1

    # Compute the difference in orientation using quaternion distance
    quat_diff = quat_mul(object_rot, quat_conjugate(goal_rot))
    orientation_error = torch.norm(quat_diff[:, 1:4], dim=1)  # Ignore the scalar part for distance
    orientation_reward = torch.exp(-orientation_temp * orientation_error)

    # Penalize excessive angular velocity
    angular_velocity_magnitude = torch.norm(object_angvel, dim=1)
    angular_velocity_penalty = torch.exp(-angular_velocity_temp * angular_velocity_magnitude)

    # Combine rewards
    total_reward = orientation_reward * angular_velocity_penalty

    # Return the total reward and individual components for debugging
    reward_components = {
        "orientation_reward": orientation_reward,
        "angular_velocity_penalty": angular_velocity_penalty
    }
    return total_reward, reward_components
```

E Detailed Results

E.1 Evaluation on Efficiency

Table 6 presents the comprehensive evaluation results across all tasks in the three benchmarks. The comparative analysis demonstrates that while achieving comparable SR or HNS to Eureka, *URDP* requires fewer simulation training episodes and LLM invocations in 92% of the experimental tasks, indicating superior sample efficiency and computational economy.

Table 6: *URDP* vs. SOTA with efficiency. *URDP* performed best in 92% of tasks in terms of NOE and NLC (bolded parts in the table).

Benchmark	Environment	Text2Reward			Eureka			*URDP*		
		HNS	NOE↓	NLC↓	HNS	NOE↓	NLC↓	HNS	NOE↓	NLC↓
Isaac	Ant	1.543	112	7	1.527	112	7	1.556	**48**	**3**
	Cartpole	1	16	1	1	16	1	1	**15**	**1**
	BallBalance	1	16	1	1	16	1	1	**16**	**1**
	Quadcopter	1.678	82	6	1.667	70	5	1.818	**41**	**2**
	FrankaCabinet	16.95	95	7	17	97	7	17.130	**57**	**4**
	Humanoid	2.305	16	1	2.306	16	1	2.646	52	3
	Anymal	1.095	87	6	1.113	91	6	1.2	**47**	**2**
	AllegroHand	2.176	121	8	2.162	95	6	2.182	**40**	**4**
	ShadowHand	1.805	111	8	1.786	105	7	1.817	**33**	**2**
		SR	NOE↓	NLC↓	SR	NOE↓	NLC↓	SR	NOE↓	NLC↓
Dexterity	BlockStack	0.67	112	7	0.67	112	7	0.68	**53**	**1**
	HandKettle	0.89	85	6	0.89	72	5	0.89	**78**	**5**
	HandDoorCloseOutward	0.96	83	6	0.9	72	5	0.97	**37**	**2**
	DoorCloseInward	1	71	5	1.0	78	5	1.0	**34**	**2**
	SwingCup	0.84	93	7	0.84	87	6	0.84	**51**	**3**
	Switch	0	58	5	0.0	58	5	0.02	76	**5**
	TwoCatchUnderarm	0	62	5	0.0	62	5	0.0	**62**	**4**
	CatchUnderarm	0.72	95	7	0.73	89	6	0.73	**67**	**4**
	CatchAbreast	0.66	88	6	0.66	83	6	0.67	**54**	**4**
	DoorOpenInward	0.04	73	5	0.04	69	5	0.06	**67**	**4**
	PushBlock	0.14	97	7	0.14	92	6	0.15	**49**	**3**
	BottleCap	0.88	110	8	0.88	96	7	0.89	**67**	**4**
	ReOrientation	0.32	75	6	0.31	66	5	0.33	**58**	**4**
	CatchOver2Underarm	0.91	77	6	0.9	74	5	0.93	**57**	**3**
	LiftUnderarm	0.89	88	6	0.89	86	6	0.89	**78**	**4**
	Over	0.92	82	6	0.92	71	5	0.92	**41**	**3**
	Pen	0.85	95	7	0.85	97	7	0.85	**78**	**4**
	DoorOpenOutward	1	76	5	1.0	75	5	1.0	**46**	**3**
	Scissors	1	76	5	1.0	77	5	1	**44**	**3**
	GraspAndPlace	0.75	93	6	0.75	85	6	0.77	**59**	**4**
		SR	NOE↓	NLC↓	SR	NOE↓	NLC↓	SR	NOE↓	NLC↓
ManiSkill2	LiftCube	0.906	112	7	0.905	96	6	0.906	**15**	**1**
	PickCube	0.879	128	8	0.884	112	7	0.885	**20**	**1**
	TurnFaucet	0.799	96	6	0.800	96	6	0.801	**34**	**2**
	OpenCabinetDoor	0.865	96	6	0.861	96	6	0.866	**31**	**2**
	OpenCabinetDrawer	0.633	112	7	0.632	96	6	0.638	**43**	**2**
	PushChair	0.657	96	6	0.654	96	6	0.657	**51**	**2**

E.2 Evaluation on the Performance of the Reward Function

Table 7 presents a comprehensive performance comparison of different methods across all tasks in three benchmarks. The results demonstrate that *URDP* consistently outperforms the baseline approaches while maintaining comparable or reduced requirements for both simulation training episodes and LLM invocations. Notably, *URDP* achieves superior performance to human-designed reward functions in 89% of the experimental tasks, highlighting the significant potential of automated reward design methodologies.

Table 7: Task-wise comparison of *URDP* with other methods. *URDP* outperforms the compared methods on 89% of the tasks.

Benchmark	Environment	Sparse	Human	Text2Reward			Eureka			*URDP*		
		HNS↑	HNS↑	HNS↑	NOE	NLC	HNS↑	NOE	NLC	HNS↑	NOE	NLC
Isaac	Ant	0	1	0.772	48	3	0.828	48	3	**1.556**	48	3
	Cartpole	0	1	1	15	1	1	15	1	**1**	15	1
	BallBalance	0	1	1	16	1	1	16	1	**1**	16	1
	Quadcopter	0	1	1.041	41	3	1.25	41	3	**1.818**	41	2
	FrankaCabinet	0	1	5.4	57	4	4.8	57	4	**17.130**	57	4
	Humanoid	0	1	2.217	52	4	2.306	52	4	**2.646**	52	3
	Anymal	0	1	0.317	47	3	0.545	47	3	**1.2**	47	2
	AllegroHand	0	1	1.196	40	3	1.594	40	3	**2.182**	40	4
	ShadowHand	0	1	1.034	33	3	1.115	33	3	**1.817**	33	2
		SR↑	SR↑	SR↑	NOE	NLC	SR↑	NOE	NLC	SR↑	NOE	NLC
Dexterity	BlockStack	0	0.69	0.11	53	4	0.12	53	4	0.679	53	1
	Kettle	0	0.02	0.89	78	5	0.89	78	5	**0.89**	72	5
	DoorCloseOutward	0.15	0.06	0.57	37	3	0.64	37	3	**0.968**	37	2
	DoorCloseInward	0	1	0.74	34	3	0.83	34	3	**1**	34	2
	SwingCup	0	0	0.62	51	4	0.53	51	4	**0.84**	51	3
	Switch	0	0	0	76	5	0.01	76	5	**0.02**	76	5
	TwoCatchUnderarm	0	0	0	62	5	0	62	5	**0**	62	4
	CatchUnderarm	0	0.51	0.58	67	5	0.63	67	5	**0.73**	67	4
	CatchAbreast	0	0.37	0.27	54	5	0.34	54	5	**0.67**	54	4
	DoorOpenInward	0	0.03	0	67	5	0	67	5	**0.06**	67	4
	PushBlock	0	0.01	0.05	49	4	0.05	49	4	**0.15**	49	3
	BottleCap	0.91	0.91	0.21	67	5	0.25	67	5	0.89	67	4
	ReOrientation	0.01	0.02	0.25	58	4	0.28	58	4	**0.33**	58	4
	CatchOver2Underarm	0	0.87	0.81	57	4	0.81	57	4	**0.93**	57	3
	LiftUnderarm	0	0.37	0.83	78	6	0.85	78	6	**0.89**	78	4
	Over	0	0.9	0.54	41	4	0.61	41	4	**0.92**	41	3
	Pen	0.01	0.74	0.67	78	6	0.63	78	6	**0.85**	78	4
	DoorOpenOutward	0.02	0.85	0.76	46	3	0.87	46	3	**1**	46	3
	Scissors	0.99	0.96	0.73	44	3	0.69	44	3	**1**	44	3
	GraspAndPlace	0	0.87	0.41	59	4	0.43	59	4	0.77	59	4
		SR↑	SR↑	SR↑	NOE	NLC	SR↑	NOE	NLC	SR↑	NOE	NLC
ManiSkill2	LiftCube	0.143	0.543	0.531	15	1	0.356	15	1	**0.906**	15	1
	PickCube	0.131	0.479	0.497	20	2	0.434	20	2	**0.885**	20	1
	TurnFaucet	0	0.598	0.631	34	3	0.516	34	3	**0.801**	34	2
	OpenCabinetDoor	0.028	0.651	0.713	31	2	0.575	31	2	**0.866**	31	2
	OpenCabinetDrawer	0	0.37	0.519	43	3	0.478	43	3	**0.64**	43	2
	PushChair	0	0.334	0.432	51	4	0.336	51	4	**0.657**	51	2

F Proofs

F.1 Determination of the Kernel Function

Theorem 1. *The kernel function equation 5 satisfied the properties of symmetry and positive semi-definiteness.*

Proof. The property of symmetry is obvious. Now we prove the positive semi-definiteness based on the properties of Matern kernel equation 3. For any given finite set of sample points $\tilde{p}^{(1)}, \tilde{p}^{(2)}, \cdots, \tilde{p}^{(n)}$, we denote the corresponding kernel matrix as

$$\tilde{K}_{ij} = \tilde{k}(\tilde{p}^{(i)}, \tilde{p}^{(j)}). \tag{9}$$

By performing coordinate scaling transformation on the sample points, we obtain new sample points

$$p^{(i)} = (\tilde{p}_1^{(i)}/l_1, \cdots, \tilde{p}_d^{(i)}/l_d),\ i = 1, 2, \cdots, n. \tag{10}$$

And the Matern kernel matrix is

$$K_{ij} = k(p^{(i)}, p^{(j)}) \tag{11}$$

Since Matern kernel equation 3 is positive semi-definite, the kernel matrix equation 11 constructed from the transformed points with Matern kernel is positive semi-definite. Furthermore, the kernel matrix of equation 5 is essentially equivalent to the Martern kernel matrix equation 11 computed on the transformed sample points, i.e.

$$r_{\text{new}}(\tilde{p}^{(i)}, \tilde{p}^{(j)}) = r(p^{(i)}, p^{(j)}) \tag{12}$$

$$\tilde{K}_{ij} = \tilde{k}(\tilde{p}^{(i)}, \tilde{p}^{(j)}) = f_\nu(r_{\text{new}}) = f_\nu(r) = K_{ij}. \tag{13}$$

Therefore, $\tilde{K}$ is positive semi-definite. □

In the newly defined weighted distance equation 4, a larger l_i in directions with more rapid variations can increase the possibility of exploration, while a smaller l_i in directions with smoother variations will reduce exploration and emphasize the exploitation of information from previously sampled points.

F.2 Convergence analysis of the Uncertainty-accelerated Expected Improvement (uEI)

The basic definitions and theorems have been defined to analyze the convergence rate of Bayesian optimization Bull (2011). Here, we briefly restate some of the key definitions required. Let $\mathcal{X} \subset \mathbb{R}^d$ be a compact set with non-empty interior. For a function $f : \mathcal{X} \to \mathbb{R}$ to be minimized, K_θ is the correlation kernel for function f prior distribution π with length-scales θ. $\mathcal{H}_\theta(\mathcal{X})$ is the reproducing-kernel Hilbert space of K_θ on $\mathcal{X}$. Let $\mathbb{P}_f^u$ and $\mathbb{E}_f^u$ denote the probability and expectation operators when minimizing the fixed function f using strategy u. The loss suffered over the ball B_R in $\mathcal{H}_\theta(\mathcal{X})$ after n steps by a strategy u is defined as,

$$L_n(u, \mathcal{H}_\theta(\mathcal{X}), R) := \sup_{\substack{f \in \mathcal{H}_\theta(\mathcal{X}) \\ \|f\|_{\mathcal{H}_\theta(\mathcal{X})} \leq R}} \mathbb{E}_f^u \left[f(x_n^*) - \min f \right] \tag{14}$$

where x_n^* is the estimated minimum of f.

It is proved that the strategy expected improvement converges at least at rate $n^{-(\nu \wedge 1)/d}$, up to logarithmic factors, where ν is the parameter in Matern kernel Bull (2011).

Theorem 2. *Assume that the function f depends only on m input variables, $m < d$, and remains constant along the other $d-m$ directions. Under such an assumption, with an appropriate choice of weighted parameters, the Uncertainty-accelerated Expected Improvement converges at least at rate $n^{-(\nu \wedge 1)/m}$, up to logarithmic factors, where ν is the parameter in Matern kernel.*

Proof. From the proof of expected improvement convergence rate Bull (2011), we observe that the parameter d in convergence rate estimation is actually the dimensionality of the sampling space. And the conclusion holds based on the condition that $\{x_n\}$ is a quasi-uniform sequence in a region of interest Narcowich et al.

(2003). Without loss of generality, let us assume that the function f depends on dimensions i_1 to i_m, and is invariant with respect to dimensions i_{m+1} to i_d. For the uEI strategy, let

$$\lambda_j = \begin{cases} 0, & j = 1, \cdots, m \\ \infty, & j = m+1, \cdots, d \end{cases} \tag{15}$$

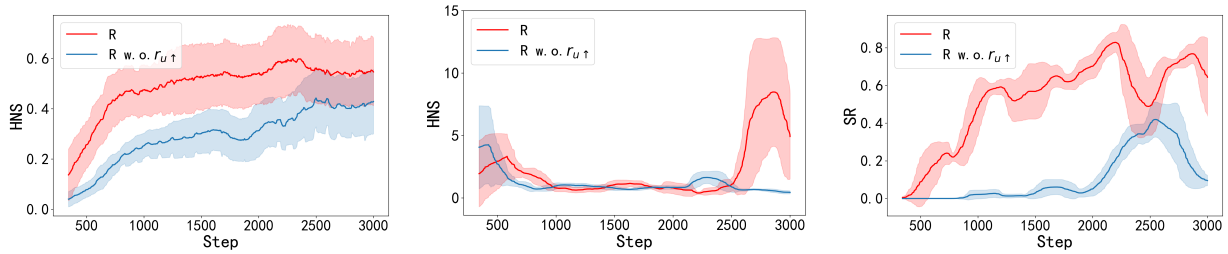
Consequently, any exploration in the directions of dimensions i_{m+1} to i_d will be discouraged. The effective dimensionality of the sampling space decreases from d to m, which leads to an improved convergence rate of at least $n^{-(\nu \wedge 1)/m}$. □

Although our theorem has focused on the limiting case in which f is entirely independent of certain directions, it illustrates how applying weighted constraints allows for dimension-specific treatment within the sampling space, thus enhancing the efficiency of the algorithm.

G Additional Analysis

G.1 Uncertainty and Reward Shaping

To validate the role of high-uncertainty reward components, we conducted ablation studies by removing these components from the reward function. Figure 8 presents comparative cases between the original reward functions (R) and its ablated counterparts (R w.o. $r_{u\uparrow}$). Our analysis reveals two key findings: (1) the removal of high-uncertainty components leads to significant performance degradation, with respective decreases of 19%, 83%, and 51% in HNS/SR metrics; and (2) reward functions retaining these components demonstrate accelerated discovery of critical states during early RL training phases, effectively reducing inefficient exploration. These results collectively demonstrate the crucial function of high-uncertainty components in both final performance and training efficiency. These results suggest that high-uncertainty reward components contribute positively to reward shaping and play an essential role in guiding effective policy learning.



(a) Anymal (b) FrankaCabinet (c) LiftUnderarm

Figure 8: The comparison between R and R w.o.$r_{u\uparrow}$ suggests that the high-uncertainty reward components ($r_{u\uparrow}$) contributes to reward shaping during the policy learning.

G.2 LLM Alternatives

URDP with Qwen2.5. In Fig. 9, we compare the performance of *URDP* with DeepSeek-v3-241226 (the results reported in the paper) and *URDP* with Qwen2.5 (qwen-max-0919) Qwen et al. (2025). These results demonstrate the consistency of the effect of *URDP* on different LLMs and eliminate concerns that the differences in the capabilities of LLMs themselves may affect the results.

H Limitation and Discussion

In this work, we investigate efficient automated reward design methodologies based on large language models (LLMs). However, constrained by inherent limitations of LLMs in spatial reasoning capabilities, our approach, like other comparable methods, faces challenges in addressing scenario-specific constraints during reward formulation. A representative case emerges in "grasping" tasks where environmental obstacles may restrict

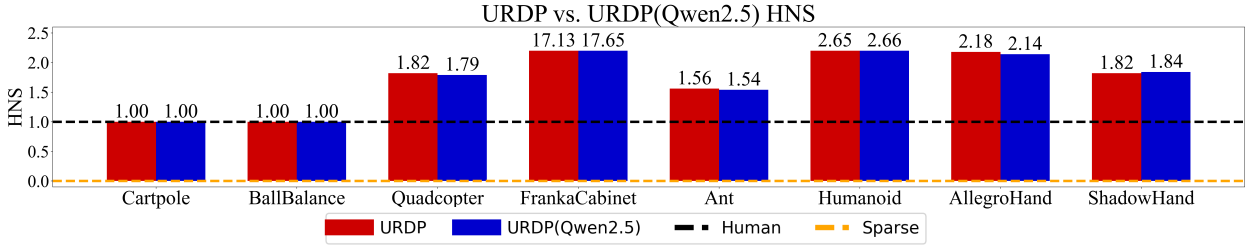


Figure 9: *URDP* demonstrates consistent performance across different LLMs.

robotic manipulation paths, constraints that should ideally be reflected in reward design. While providing detailed environmental descriptions in prompts may partially mitigate this issue, a more fundamental solution would involve integrating video-language models (VLMs) into the reward design framework. VLMs demonstrate superior spatial perception capabilities that could enrich the understanding of RL task objectives, environmental constraints, and reward composition. Nevertheless, incorporating VLMs introduces new challenges regarding computational scalability during reward design and tuning processes. We therefore identify this as a critical yet underexplored research direction worthy of systematic investigation.

To maintain simplicity in presenting our work, we employ the base capabilities of large language models without sophisticated inference-time enhancement techniques (e.g., chain-of-thought, test-time training). However, advanced reasoning techniques have demonstrated significant improvements in handling complex logical tasks, as evidenced in code generation and mathematical reasoning domains. We posit these methods would similarly enhance reward function code design. Ultimately, substantial exploration potential remains for large language model techniques in automated reward design.