Large Language Models as Efficient Reward Function Searchers for Custom-Environment Multi-Objective Reinforcement Learning

Guanwen Xie*, Jingzehua Xu*, Yiyuan Yang†, Yimian Ding*, Shuai Zhang‡
*MicroMasters Program in Statistics and Data Science, Massachusetts Institute of Technology, USA
†Department of Computer Science, University of Oxford, United Kingdom
‡Department of Data Science, New Jersey Institute of Technology, USA
Email: sz457@njit.edu

***Abstract*—Leveraging large language models (LLMs) for designing reward functions demonstrates significant potential. However, achieving effective design and improvement of reward functions in reinforcement learning (RL) tasks with complex custom environments and multiple requirements presents considerable challenges. In this paper, we enable LLMs to be effective white-box searchers, highlighting their advanced semantic understanding capabilities. Specifically, we generate reward components for each explicit user requirement and employ the reward critic to identify the correct code form. Then, LLMs assign weights to the reward components to balance their values and iteratively search and optimize these weights based on the context provided by the training log analyzer, while adaptively determining the search step size. We applied the framework to an underwater data collection RL task without direct human feedback or reward examples (zero-shot). The reward critic successfully correct the reward code with only one feedback for each requirement, effectively preventing unrectifiable errors that can occur when reward function feedback is provided in aggregate. The effective initialization of weights enables the acquisition of different reward functions within the Pareto solution set without weight search. Even in the case where a weight is 100 times off, on overage fewer than four iterations are needed to obtain solutions that meet user requirements. The framework also works well with most prompts utilizing GPT-3.5 Turbo, since it does not require advanced numerical understanding or calculation.**

***Index Terms*—Large language models, Multi-objective reinforcement learning, Reward function design**

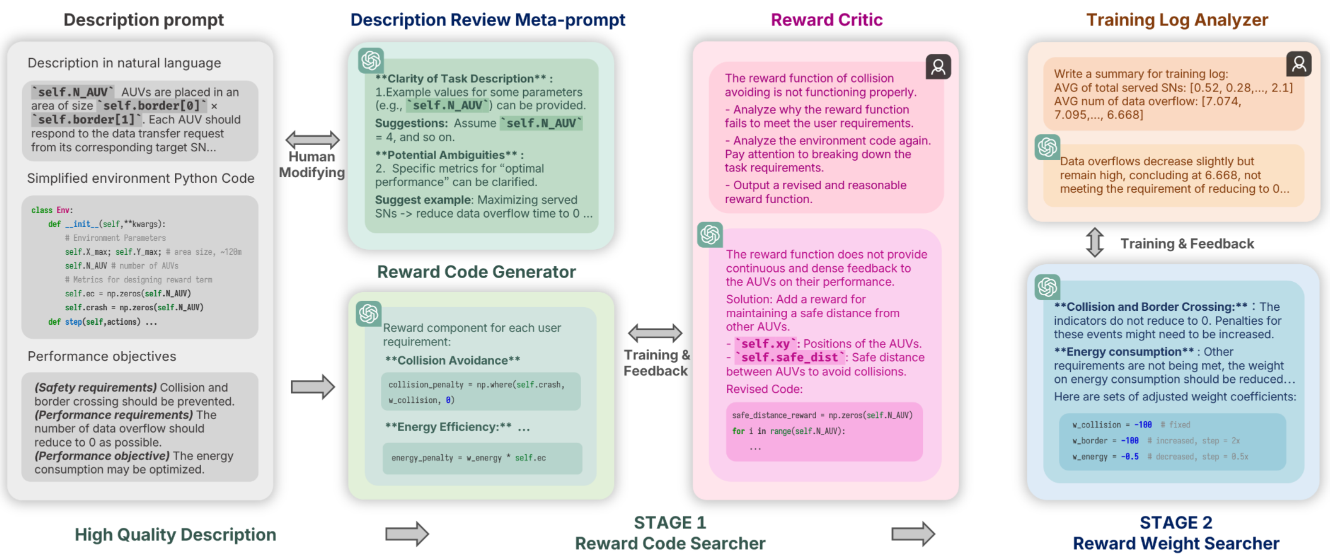
I. Introduction

Given their robust abilities to address user requirements, reinforcement learning (RL) techniques are being increasingly utilized for intricate, multi-objective tasks. Nonetheless, as the variety and quantity of requirements and optimization goals grow, the design of reward functions becomes more complex, necessitating significant effort to adjust the structure and coefficients of each reward component. This complexity is further compounded by the fact that researchers' needs often fluctuate with different scenarios and over time, and can sometimes be ambiguous [1], thereby posing a considerable challenge to achieving optimal performance.

Large language models (LLMs) are trained on extensive text data [2], enabling them to demonstrate strong problem-solving and content generation abilities when given well-crafted prompts, even without prior domain knowledge. They can also achieve self-improvement through human feedback. The application of LLMs in creating functional code has shown remarkable performance across various tasks, such as dexterous robots control [3]–[5] and Minecraft playing [6], [7], highlighting their potential in zero-shot scenarios with limited self-evolution iterations. However, this improvement process depends on trial-and-error exploration. When there is a single, clear objective (e.g., success rate), the search space for designing functions and tuning parameters is confined, allowing for gradual optimization through iterations. However, for complex reward functions, where reward components and their weights are determined simultaneously, problems like incorrect code and imbalanced weights may emerge, which are challenging to resolve purely through training feedback. An analogous topic is LLM-driven white-box optimization [2], [8], [9], such as hyperparameter optimization (HPO) [8], [10]. This approach involves abstracting all explicitly defined coefficients and the function code itself into parameters, allowing LLMs to perform comprehensive analyses and improvements on well-defined machine learning tasks. This paradigm aligns with the observation that LLMs are particularly adept at summarizing and heuristically generating code within specific and clear task contexts, yet they are less proficient in addressing black-box optimization problems [11]. These considerations underscore the necessity of providing clear and precise task descriptions while also rationally exploring the extensive search space.

Based on above analysis, in this paper we utilize task decomposition and white-box search, employing LLMs as effective searchers to fully leverage its semantic understanding capabilities. Unlike previous work on LLM-aided reward function design, we separate the process into two stages: reward code design and weight assignment, while breaking down the multi-objective RL task into multiple explicit numerical goals. This division helps to eliminate ambiguity in training feedback. Throughout both stages, we implement a clear feedback and self-evolution search paradigm, using a reward

The full-text prompts, examples of LLM-generated answers, and source code are available at https://360zmem.github.io/LLMRsearcher/ .



Description in natural language

`self.N_AUV` AUVs are placed in an area of size `self.border[0]` × `self.border[1]`. Each AUV should respond to the data transfer request from its corresponding target SN...

Simplified environment Python Code

```
class Env:
    def __init__(self,**kwargs):
        # Environment Parameters
        self.X_max; self.Y_max; # area size, ~120m
        self.N_AUV # number of AUVs
        # Metrics for designing reward term
        self.ec = np.zeros(self.N_AUV)
        self.crash = np.zeros(self.N_AUV)
    def step(self,actions) ...
```

Performance objectives

(Safety requirements) Collision and border crossing should be prevented. ***(Performance requirements)*** The number of data overflow should reduce to 0 as possible. ***(Performance objective)*** The energy consumption may be optimized.

****Clarity of Task Description**** :
1.Example values for some parameters (e.g., `self.N_AUV`) can be provided.
Suggestions: Assume `self.N_AUV` = 4, and so on.
****Potential Ambiguities**** :
2. Specific metrics for "optimal performance" can be clarified.
Suggest example: Maximizing served SNs -> reduce data overflow time to 0 ...

Reward component for each user requirement:
****Collision Avoidance****

```
collision_penalty = np.where(self.crash,
w_collision, 0)
```

****Energy Efficiency:**** ...

```
energy_penalty = w_energy * self.ec
```

The reward function of collision avoiding is not functioning properly.
- Analyze why the reward function fails to meet the user requirements.
- Analyze the environment code again. Pay attention to breaking down the task requirements.
- Output a revised and reasonable reward function.

The reward function does not provide continuous and dense feedback to the AUVs on their performance.
Solution: Add a reward for maintaining a safe distance from other AUVs.
- `self.xy`: Positions of the AUVs.
- `self.safe_dist`: Safe distance between AUVs to avoid collisions.
Revised Code:

```
safe_distance_reward = np.zeros(self.N_AUV)
for i in range(self.N_AUV):
    ...
```

Write a summary for training log:
AVG of total served SNs: [0.52, 0.28,..., 2.1]
AVG num of data overflow: [7.074, 7.095,..., 6.668]

Data overflows decrease slightly but remain high, concluding at 6.668, not meeting the requirement of reducing to 0...

****Collision and Border Crossing:**** : The indicators do not reduce to 0. Penalties for these events might need to be increased.
****Energy consumption**** : Other requirements are not being met, the weight on energy consumption should be reduced...
Here are sets of adjusted weight coefficients:

```
w_collision = -100  # fixed
w_border = -100  # increased, step = 2x
w_energy = -0.5  # decreased, step = 0.5x
```

Fig. 1. The main architecture and prompt examples of the LLM-based reward search framework.

critic to refine reward components for each user requirement and optimizing the weight coefficients of these components based on training feedback. By employing a training log analyzer, we separate numerical and textual context, thereby enhancing the accuracy and flexibility of weight searching. We apply this framework to the reward functions design for an underwater data collection task from our previous research [12], achieving the goal with minimum iterations of feedback and weight searches. Notably, aside from the task description, our framework operates without direct human feedback (easy for writing an automatic script with format constraints for LLM outputting), but it can also be effectively combined with human knowledge, for instance, by using weight searchers to optimize the weights and values of human-designed reward functions. The main architecture and important prompt examples are illustrated in Fig. 1.

II. Methodology

A. High Quality Environment Description

Task description is a common component of most subsequent prompts, comprising text descriptions, environment code, or APIs, which include variables and functions essential for designing the reward function, as well as user requirements. We decompose the user requirements into numerical-clear performance demands (e.g., obstacle avoidance to achieve zero collision). Ambiguous task descriptions may hinder LLMs' ability to generate correct reward functions. To address this, we design a meta-prompt to provide users with suggestions for enhancing the quality of the description. This meta-prompt enables LLMs to identify potential issues within the prompt, such as unclear structural organization and a lack of necessary information or explanations. We instruct LLMs to output in a fixed format to maximize the identification of potential issues.

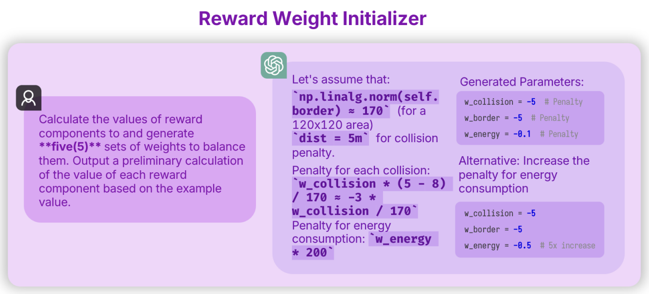
B. Reward Code Generator

The code generation process can utilize existing LLM-aided reward function design frameworks. However, instead of creating a single reward function for the entire output, we generate individual reward components tailored to each specific user requirement. We also task LLMs with outputting explanations for the generated reward components, which may improve the correctness of the generated code. Despite this, the initial code produced by LLMs is likely to be incorrect due to the absence of prior knowledge about custom environments and the complexity of lengthy contexts. Therefore, we test each component separately and correct errors using a LLM-based reward critic. The reward critic follows instructions from a step-by-step guide, namely first listing possible reasons for code failure, then reviewing the environment code and the requirement, and finally outputting the correct function code. This process allows LLMs to clearly analyze errors within the reward function, thereby avoiding the ambiguity of overall feedback. In addition, if the variables and functions are insufficient to write new components, LLMs are allowed to fabricate relative variables and prompt the user to complete them, which effectively overcome the negative effects caused by incomplete environment descriptions.

C. Reward Weight Searcher

Multi-objective RL necessitates not only the proper form of reward components but also the accurate scaling of these components. We utilize LLMs as effective weight searchers under explicit task contexts.

Appropriate weights searching necessitates an effective starting point. We leverage specialized instructions that require LLMs to pre-calculate approximate values of the components and adjust the weights to make values of these components have the same scale, This approach prevents the initial weight



Calculate the values of reward components to and generate **five(5)** sets of weights to balance them. Output a preliminary calculation of the value of each reward component based on the example value.

Let's assume that:
`np.linalg.norm(self.border) ≈ 170` (for a 120x120 area)
`dist = 5m` for collision penalty.
Penalty for each collision: `w_collision * (5 - 8) / 170 ≈ -3 * w_collision / 170`
Penalty for energy consumption: `w_energy * 200`

Generated Parameters:

```
w_collision = -5  # Penalty
w_border = -5  # Penalty
w_energy = -0.1  # Penalty
```

Alternative: Increase the penalty for energy consumption

```
w_collision = -5
w_border = -5
w_energy = -0.5  # 5x increase
```

Fig. 2. Prompt examples of the reward weight initializer.

groups from deviating significantly from the optimal solutions. We refer to this process as the reward weight initializer, with sample prompts illustrated in Fig. 2.

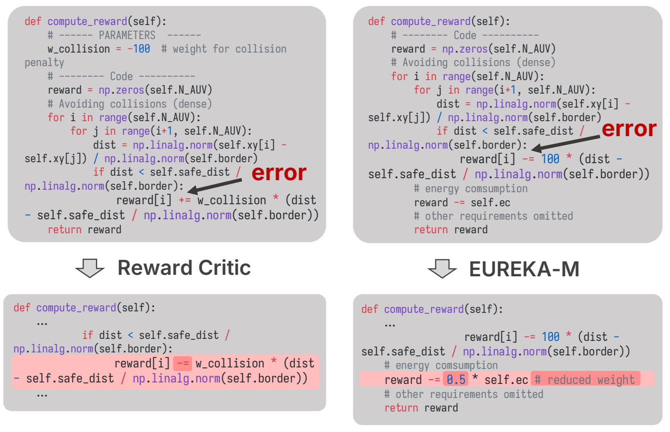
Afterward, LLMs offer weight adjustment suggestions based on the training results. Existing methods often utilize a Python list-style training log to present these results. However, due to LLMs' limited numerical comprehension, handling multiple performance indicators can make the prompt excessively lengthy and complex, potentially leading to inaccurate suggestions generation. Consequently, we employ a training log analyzer to produce a concise task summary context. Utilizing these summaries, the reward weight searcher delivers specific weight adjustment suggestions, such as whether to increase, decrease, or maintain a given weight. To accelerate the search process, we instruct LLMs to determine the search step size. For instance, if an increase in weight still fails to meet certain user requirements, it may indicate that the weight value is significantly deviated and requires a larger step size. Conversely, smaller step sizes might be needed in the later stages of the search. Additionally, the weight generation process creates K=5 sets of weights, each of which adopts different search strategies, prioritizing different user requirements.

III. Experiments

A. Task Description and Parameters

To evaluate the proposed framework, we select the underwater data collection task from our previous work [12], which utilizes the RL algorithm to control multiple autonomous underwater vehicles (AUVs) for data collection. We task LLMs with designing reward functions encompassing safety requirements (collision and border crossing avoidance), performance requirements (timely serving of the target SN to minimize data overflow), and performance objectives (reduce energy consumption), without providing any reward examples (namely zero-shot), as shown in the task description in Fig. 1. We refer to the original paper for the system models and simulation parameters. For determinacy, TD3 [13] is used as the RL algorithm instead of MAISAC, as in the original paper.

For experiments, we utilize `gpt-4o-2024-08-06` (denoted as **GPT-4o**) as the default LLM due to its improved performance and reasonable API pricing. We also conduct experiments using the cost-efficient smaller version of OpenAI's



```
def compute_reward(self):
    # ------ PARAMETERS  ------
    w_collision = -100  # weight for collision
penalty
    # -------- Code ----------
    reward = np.zeros(self.N_AUV)
    # Avoiding collisions (dense)
    for i in range(self.N_AUV):
        for j in range(i+1, self.N_AUV):
            dist = np.linalg.norm(self.xy[i] -
self.xy[j]) / np.linalg.norm(self.border)
            if dist < self.safe_dist /
np.linalg.norm(self.border):
                reward[i] += w_collision * (dist
- self.safe_dist / np.linalg.norm(self.border))
    return reward
```

```
def compute_reward(self):
    # -------- Code ----------
    reward = np.zeros(self.N_AUV)
    # Avoiding collisions (dense)
    for i in range(self.N_AUV):
        for j in range(i+1, self.N_AUV):
            dist = np.linalg.norm(self.xy[i] -
self.xy[j]) / np.linalg.norm(self.border)
            if dist < self.safe_dist /
np.linalg.norm(self.border):
                reward[i] -= 100 * (dist -
self.safe_dist / np.linalg.norm(self.border))
        # energy comsumption
        reward -= self.ec
        # other requirements omitted
        return reward
```

```
def compute_reward(self):
    …
            if dist < self.safe_dist /
np.linalg.norm(self.border):
                reward[i] -= w_collision * (dist
- self.safe_dist / np.linalg.norm(self.border))
    …
```

```
def compute_reward(self):
    …
                reward[i] -= 100 * (dist -
self.safe_dist / np.linalg.norm(self.border))
    # energy comsumption
    reward -= 0.5 * self.ec # reduced weight
    # other requirements omitted
    return reward
```

Fig. 3. When the collision reward term is reversed, the reward critic corrects the error, but directly feeding back the entire reward function only results in weight modifications.

LLM, `gpt-4o-mini-2024-07-18` (denoted as **GPT-4om**). The LLM parameters are set to `temperature=0.5` and `Top P=1`.

Meanwhile, we design a baseline that takes the reward functions and numerical values as a whole. This approach is similar to Eureka [3]; however, during the reward function revision stage, it processes one or more reward functions along with their training logs summarized by the training log analyzer as inputs, and generates K=5 outputs simultaneously. This approach remains consistent with the reward weight searcher, in contrast to generating a single reward function repeatedly in an i.i.d. manner as in Eureka. We denote this baseline as **EUREKA-M**.

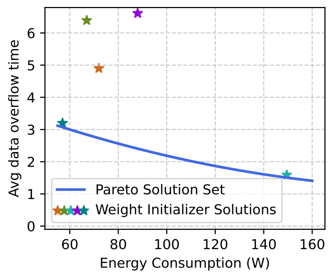
B. Main Results and Case Studies

Reward Critic can generate correct code stably and quickly. For reward code generator, due to the long context and possibly unclear task descriptions, the generated code may not be entirely correct. For example, we find that the initially generated reward functions may have sparse reward terms, miscalculate distances between AUVs and the boundary, contain symbolic errors, among other issues. To address these issues, the reward critic generate feedback and revised code for each component. We test each component separately and utilize a variation of the training log analyzer, which outputs `[YES]` and `[NO]` to indicate whether this component fulfills the requirement. We find that the reward critic can effectively detect various errors and then rewrite reward components based on description and variables of the environment class. For each component, it only needs to be once to obtain the correct feedback results.

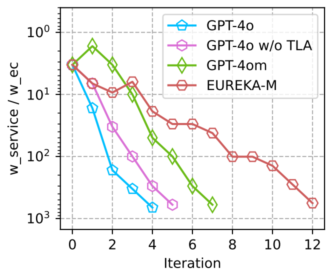
To investigate the advantages of the reward critic, we manually introduce an elusive error into the reward function by reversing the penalty term for collision into a reward term, namely reversing the symbol `reward -= ...` to `reward += ...`. Example outputs of the reward critic and EUREKA-M is shown in Fig. 3. The reward critic can identify the error, but EUREKA-M fails to do so, only modifying reward weights, even when prompted with "This code contains

TABLE I
THE NUMBER OF ITERATIONS OF SEARCHING WEIGHTS UNDER DIFFERENT EXPERIMENT SETTINGS. THE EXPERIMENT ARE PERFORMED 5 TIMES, AND THE MEAN VALUES AND STANDARD DEVIATIONS ARE REPORTED.

Setting	**GPT-4o**	**GPT-4o w/o TLA**	**GPT-4om**
RWI initialized	**0.40±0.49**		
RWI w/o balance	1.20±0.75	1.60±1.02	1.80±1.17
100x off	4.80±1.17	6.00±1.78	8.60±1.74



(a) Solutions generated from the reward weight initializer.



(b) Change of step sizes under different settings during iteration.

Fig. 4. Figures of reward weight searching. (a) Solutions generated from the reward weight initializer. (b) Change of step sizes under different settings during iteration.

errors." We also attempted to use the sparse term-only reward function for EUREKA-M to modify, but unless explicitly prompted to generate a dense term, it only continues with weight modification. This suggests that the reward design process based on training feedback may be ineffective without explicit human input.

Reward weight initialization and search. We obtain initial groups of weights from the reward weight initializer, and the energy consumption and average data overflow times (lower is better) of these solutions are illustrated in Fig. 4(a). Although three groups of generated weights do not meet user requirements, two groups (emphasizing timely response to target SNs and energy consumption, respectively) successfully achieve Pareto solutions. This means that no further search is required, or only a more refined search may be necessary for a specific point of the Pareto set. Then, we employ the reward weight searcher to iteratively adjust the weight coefficients (denoted as **RWI initialized**). We also perform ablation experiments by removing the value balance process of the weight initializer (i.e., eliminating the example values from the environment description and removing prompts requiring LLMs balance reward values, denoted as **RWI w/o balance**), and the training log analyzer (denoted as **GPT-4o w/o TLA**). Additionally, we task LLMs to search from a weight group generated from **RWI initialized**, but with the weight of the energy consumption penalty term increased by a factor of 100 (denoted as **100x off**), to understand the search details. Table 1 displays the number of iterations required to meet user demands under different settings, while Fig. 4(b) depicts the search step sizes during iteration. When the weight initialization does not consider the balance between reward components, the ratio between weights compared to RWI-initialized settings could differ by a factor of 1 to 50, while leading to a significant increase in the deviation of generated weights. Nevertheless, since the distance is not substantial, only 0-2 iterations are necessary to find a feasible solution.

For the **100x off** experiment group, **GPT-4o** achieves minimum search iterations. Specifically, after one iteration, **GPT-4o** recognizes that the weight setting may be far from a feasible solution and thus tries to increase the search step size. However, when the training log analyzer is removed, the search process becomes less flexible. We also conduct the feedback evolution process using EUREKA-M, and the step size is lower than that of **GPT-4o w/o TLA**, and the modification suggestions sometimes contain errors.

The difference between utilizing GPT-4o and GPT-4om. Intuitively, GPT-4om and open-source LLMs exhibit weaker overall reasoning abilities compared to GPT-4o, and their numerical analysis and mathematical capabilities are also comparatively limited [14], which leads to poor performance in designing reward functions for robotic control [15]. This results in drawbacks for utilizing GPT-4om in our framework, such as the reward function initializer not functions. In terms of answering content-generating prompts (such as code design and training log analysis), the quality of GPT-4om is slightly inferior to that of GPT-4o. Despite this, GPT-4om performs adequately. Similar to GPT-4o, the reward critic only needs one feedback to correct the code per requirement, while the reward weight searcher can generate accurate modification suggestions and surpass the performance of EUREKA-M, due to clear task definitions and textual feedback. However, as shown in Fig. 4(b), due to its limited capabilities, the step sizes provided by GPT-4om lack flexibility, necessitating more search iterations. It is worth noting that GPT-4om's output sometimes overlooks certain requirements or experiences format degradation in extended contexts, necessitating stricter constraints on the output format specified in the input prompt.

IV. Conclusion and Discussion

In this paper, we decompose a multi-objective task into clear user requirements, enabling LLMs to function as zero-shot searchers that receive clear feedback and effectively generate reward functions. LLMs are tasked to reward components, which are subsequently corrected by the reward critic to prevent potential errors. Leveraging the enhanced numerical calculation capabilities of the latest GPT-4o, we initialize weights that balance the value of reward components, allowing us to obtain feasible Pareto solutions without the need for extensive searching. Furthermore, based on search history and non-numerical contexts provided by the training log analyzer, GPT-4o can flexibly adopt different search strategies and step sizes, thereby accelerating the search process. In addition, with the exception of the reward weight initializer, most prompts exhibit acceptable performance in GPT-4om. Future work may focus on developing clearer and more automated task descriptions and verifying the LLM-aided reward design process across a broader range of tasks.

References

[1] C. F. Hayes, R. Rădulescu, E. Bargiacchi, J. Källström, M. Macfarlane, M. Reymond, T. Verstraeten, L. M. Zintgraf, R. Dazeley, F. Heintz *et al.*, "A practical guide to multi-objective reinforcement learning and planning," *Autonomous Agents and Multi-Agent Systems*, vol. 36, no. 1, p. 26, 2022.

[2] S. Liu, C. Chen, X. Qu, K. Tang, and Y.-S. Ong, "Large language models as evolutionary optimizers," *arXiv preprint arXiv:2310.19046*, 2023.

[3] Y. J. Ma, W. Liang, G. Wang, D.-A. Huang, O. Bastani, D. Jayaraman, Y. Zhu, L. Fan, and A. Anandkumar, "Eureka: Human-level reward design via coding large language models," in *The Twelfth International Conference on Learning Representations*, 2024.

[4] Y. Zeng, Y. Mu, and L. Shao, "Learning reward for robot skills using large language models via self-alignment," *arXiv preprint arXiv:2405.07162*, 2024.

[5] W. Yu, N. Gileadi, C. Fu, S. Kirmani, K.-H. Lee, M. G. Arenas, H.-T. L. Chiang, T. Erez, L. Hasenclever, J. Humplik *et al.*, "Language to rewards for robotic skill synthesis," *arXiv preprint arXiv:2306.08647*, 2023.

[6] Y. Wu, S. Y. Min, S. Prabhumoye, Y. Bisk, R. R. Salakhutdinov, A. Azaria, T. M. Mitchell, and Y. Li, "Spring: Studying papers and reasoning to play games," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[7] H. Li, X. Yang, Z. Wang, X. Zhu, J. Zhou, Y. Qiao, X. Wang, H. Li, L. Lu, and J. Dai, "Auto mc-reward: Automated dense reward design with large language models for minecraft," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024.

[8] M. Zhang, N. Desai, J. Bae, J. Lorraine, and J. Ba, "Using large language models for hyperparameter optimization," in *NeurIPS 2023 Foundation Models for Decision Making Workshop*, 2023.

[9] Z. Ma, H. Guo, J. Chen, G. Peng, Z. Cao, Y. Ma, and Y. jiao Gong, "Llamoco: Instruction tuning of large language models for optimization code generation," *arXiv preprint arXiv:2403.01131*, 2024.

[10] C. Wang, X. Liu, and A. H. Awadallah, "Cost-effective hyperparameter optimization for large language model generation inference," in *International Conference on Automated Machine Learning*. PMLR, 2023, pp. 21–1.

[11] B. Huang, X. Wu, Y. Zhou, J. Wu, L. Feng, R. Cheng, and K. C. Tan, "Exploring the true potential: Evaluating the black-box optimization capability of large language models," *arXiv preprint arXiv:2404.06290*, 2024.

[12] Z. Zhang, J. Xu, G. Xie, J. Wang, Z. Han, and Y. Ren, "Environment- and energy-aware auv-assisted data collection for the internet of underwater things," *IEEE Internet of Things Journal*, vol. 11, no. 15, pp. 26 406–26 418, 2024.

[13] S. Fujimoto, H. Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," in *International conference on machine learning*. PMLR, 2018, pp. 1587–1596.

[14] H. Chen, F. Jiao, X. Li, C. Qin, M. Ravaut, R. Zhao, C. Xiong, and S. Joty, "Chatgpt's one-year anniversary: are open-source large language models catching up?" *arXiv preprint arXiv:2311.16989*, 2023.

[15] T. Xie, S. Zhao, C. H. Wu, Y. Liu, Q. Luo, V. Zhong, Y. Yang, and T. Yu, "Text2reward: Reward shaping with language models for reinforcement learning," in *The Twelfth International Conference on Learning Representations*, 2024.