Accelerating Multi-Agent Reinforcement Learning with LLM-Generated Reward Patches

Bowen Lv
School of Computer Science and Technology
Huazhong University of Science and Technology
Wuhan, China
ORCID: 0009-0004-4085-7321

***Abstract*—We propose a novel framework that leverages Large Language Models (LLMs) to automate reward engineering for Multi-Agent Reinforcement Learning (MARL). Instead of generating code or end-to-end policies, an LLM produces a structured and human-readable *reward patch* that specifies shaping rewards, penalties, and success conditions. This patch is then applied to a pre-trained baseline policy during fine-tuning, effectively decoupling reward logic from environment and agent code while enhancing modularity and interpretability.**

In the PettingZoo `simple_spread` environment, our method achieves a significant average improvement of 3.6× in sample efficiency compared to training with sparse rewards, with the best run accelerating learning by up to 46×. An ablation study highlights that these gains arise from the synergy between foundational skills of the pre-trained policy and the task-specific guidance of the LLM-generated patch.

This work demonstrates a practical and efficient paradigm for integrating generative AI into MARL, providing new insights into cooperative multi-agent learning and reward design.

***Index Terms*—Multi-Agent Reinforcement Learning, Reward Engineering, Large Language Models, Sample Efficiency, Transfer Learning.**

I. Introduction

Multi-Agent Reinforcement Learning (MARL) provides a powerful paradigm for solving coordination problems in domains such as robotic swarms, autonomous driving, and resource management [1], [2]. A major obstacle to practical deployment, however, lies in the design of reward functions. Sparse rewards—where agents receive feedback only upon task completion—lead to extremely poor sample efficiency, as agents rarely discover effective strategies through random exploration. Manual reward shaping is often required, but it is labor-intensive, brittle to task variations, and demands deep domain expertise [3], [4].

Recent advances in Large Language Models (LLMs) have demonstrated impressive capabilities in code generation, reasoning, and planning, motivating their application to reinforcement learning pipelines [5], [6]. Prior work has explored LLMs as policy generators, planners, or even environment models. While promising, these approaches often lack interpretability and may require substantial computational resources.

In this paper, we introduce an alternative paradigm: **LLM-assisted reward engineering**. Instead of generating executable code or end-to-end policies, we prompt an LLM to produce a *structured reward patch*—a simple, human-readable configuration file (e.g., JSON) that specifies penalties, shaping rewards, and success conditions. This patch is applied during fine-tuning to a pre-trained baseline policy, effectively decoupling reward logic from agent and environment code.

Our framework offers three main advantages:

- **Modularity and Decoupling:** Reward design is independent of agent implementation, enabling rapid iteration without modifying core components.
- **Interpretability:** The patch provides explicit and transparent reward definitions, which can be easily inspected and refined by human engineers.
- **Efficiency:** Combining a pre-trained policy with dense LLM-generated guidance significantly improves sample efficiency and accelerates adaptation to new tasks.

We validate our approach on the PettingZoo `simple_spread` coordination task. Fine-tuning with an LLM-generated reward patch improves sample efficiency by an average of 3.6× compared to sparse-reward training, with best-case performance reaching a 46× acceleration. Ablation studies further highlight the synergy between pre-trained foundations and LLM-generated guidance. These results demonstrate a practical and scalable path for integrating generative AI into cooperative multi-agent learning.

II. Methodology

Our proposed framework consists of three main components: (1) a baseline policy pre-trained on a foundational task, (2) a structured reward patch generated by an LLM, and (3) an event-driven mechanism that applies the patch during a fine-tuning phase. The overall workflow is depicted in Fig. 1.

A. Structured Reward Patch

The core of our method is the structured reward patch, a configuration file that declaratively defines reward modifications. We use the JSON format for its simplicity and readability. The patch is organized into three primary sections:

- **`penalties`**: Defines fixed negative rewards for undesirable events (e.g., collisions, entering a hazard zone).
- **`shaping`**: Defines rewards or penalties that are scaled by numerical values from the environment. This is used

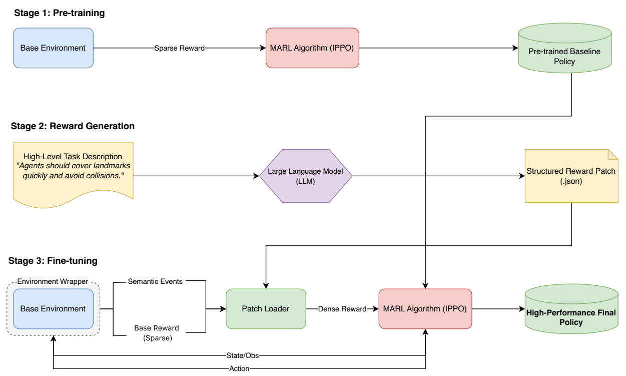
Stage 1: Pre-training
Base Environment
Sparse Reward
MARL Algorithm (IPPO)
Pre-trained Baseline Policy
Stage 2: Reward Generation
High-Level Task Description
"Agents should cover landmarks quickly and avoid collisions."
Large Language Model (LLM)
Structured Reward Patch (.json)
Stage 3: Fine-tuning
Environment Wrapper
Base Environment
Semantic Events
Base Reward (Sparse)
Patch Loader
Dense Reward
MARL Algorithm (IPPO)
High-Performance Final Policy
State/Obs
Action

Fig. 1. Overall framework. A baseline policy is pre-trained with a sparse reward. An LLM generates a reward patch from high-level instructions. The patch is then used to fine-tune the baseline policy on the target task, guided by dense rewards from an event-driven wrapper.

to create dense gradients, such as rewarding an agent for reducing its distance to a target.

- **success**: Defines fixed positive rewards for achieving key objectives (e.g., covering a landmark, completing the entire task).

This structure allows an LLM to translate high-level instructions like ”agents should avoid collisions and cover landmarks quickly” into a concrete set of machine-readable rules. Fig. 1 shows an example of such a patch.

```
{
  "penalties": {
    "enter_hazard": -1.0,
    "collision": -0.2
  },
  "shaping": {
    "agent0_centroid_bonus": 0.05,
    "time_progress_bonus": 0.01
  },
  "success": {
    "landmark_coverage": 1.0
  }
}
```

Listing 1. An example of a structured reward patch generated by the LLM. It defines penalties, shaping rewards, and success conditions in a human-readable JSON format.

B. Event-Driven Reward Application

To apply the reward patch, we introduce a lightweight environment wrapper. This wrapper observes the underlying state of the environment at each timestep and generates a dictionary of rich, semantic events. These events can be boolean flags (e.g., approach_landmark_close = true) or numerical values (e.g., progress_vs_last_step = 0.1).

A `PatchLoader` module reads the reward patch and, at each step, calculates the final reward for each agent by combining the base environment reward with the modifications specified in the patch, as triggered by the events from the wrapper. The final reward R_t for an agent at timestep t is computed as shown in Eq. (1):

$$R_t = R_{base} + \sum_{e \in \mathcal{E}_{bool}} w_e \cdot \mathbb{I}(e) + \sum_{v \in \mathcal{E}_{num}} w_v \cdot v \tag{1}$$

where R_{base} is the original environment reward, $\mathcal{E}_{bool}$ and $\mathcal{E}_{num}$ are the sets of boolean and numerical events, $\mathbb{I}(e)$ is an indicator function, and w_e, w_v are the corresponding weights from the reward patch.

C. Pre-training and Fine-tuning

Our training process occurs in two stages:

1) **Pre-training**: We first train a baseline policy using a standard MARL algorithm (e.g., Independent Proximal Policy Optimization (IPPO) [7]) with only the sparse, terminal reward from the original environment. This allows the agents to learn fundamental skills, such as basic movement and collision avoidance, without being biased by a specific task.
2) **Fine-tuning**: We then load the weights of the pre-trained policy and continue training (fine-tuning) on the target task. During this phase, the reward patch is applied, providing dense, task-specific signals that guide the agents to an effective solution rapidly.

This two-stage process leverages transfer learning to maximize sample efficiency.

III. Experiments

A. Experimental Setup

extbfEnvironment: We use the `simple_spread` environment from the PettingZoo library [1], which features 3 agents tasked with cooperatively covering 3 landmarks. The environment provides a sparse reward only when all landmarks are covered.

Evaluation Metric: The default reward is sparse and noisy. To better measure task performance, we use the **mean minimum distance**, defined as the average of each agent's Euclidean distance to its nearest uncovered landmark. A lower value indicates better performance. We use an Exponential Moving Average (EMA) of this metric for smoother evaluation.

Compared Methods: We compare three different training strategies:

1) **Baseline**: Agents are trained from scratch using only the sparse environment reward.
2) **Scratch (Patched)**: Agents are trained from scratch using the dense rewards from our reward patch. This is an ablation to measure the impact of the patch alone.
3) **Adapt (Pre-trained + Patch)**: Our full proposed method. Agents are initialized from the pre-trained Baseline model and fine-tuned using the reward patch.

All methods use the same IPPO implementation and hyperparameters for a fair comparison.

B. Results and Analysis

Fig. 2 presents the learning curves for the three methods. The results clearly demonstrate the effectiveness of our approach.

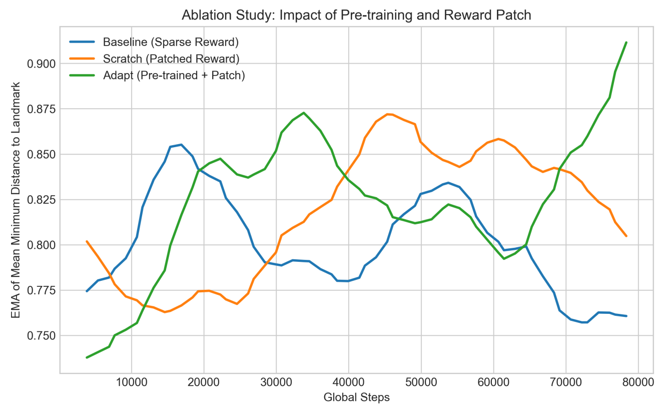

Fig. 2. Ablation study results. Our full method, Adapt (Pre-trained + Patch), significantly outperforms both the Baseline and training from scratch with the patch, demonstrating the synergistic effect of pre-training and reward patching.

Quantitative Analysis: We measure the sample efficiency by the number of environment steps required to reach a performance threshold of ema_mean_min_dist ≤ 0.75. The results are summarized in Table I. The Adapt method reaches the threshold in an average of **19,456 steps**, representing a **3.6-fold improvement** over the Baseline, which required **70,656 steps**. While this average improvement is significant, the best-case run for the Adapt method reached the threshold in just **1,536 steps** (a 46-fold improvement), highlighting the method's high potential but also indicating variability in performance across runs.

TABLE I
SAMPLE EFFICIENCY COMPARISON

Method	**Steps to reach threshold** (≤ 0.75)
Baseline (Sparse Reward)	70,656
Scratch (Patched Reward)	$> 100,000$
Adapt (Pre-trained + Patch)	**19,456 (avg.)**

Ablation Study Insights: The performance of the Scratch (Patched) method provides a crucial insight. While it learns faster than the Baseline initially, it fails to converge to a strong policy and never consistently reaches the performance threshold. This indicates that while the dense reward patch provides essential guidance, it is not sufficient on its own. The agents appear to get stuck in a local optimum.

In contrast, the Adapt method, which starts from a pre-trained policy, leverages the foundational knowledge of basic coordination and immediately utilizes the patch's guidance to solve the task efficiently. This confirms our hypothesis: the remarkable performance gain is due to the **synergy** between a solid pre-trained foundation and the sharp, task-specific guidance of the LLM-generated reward patch.

IV. Related Work

Our work intersects with several research areas, including reward shaping, multi-agent reinforcement learning, and the application of LLMs to sequential decision-making.

Classical reward shaping, formally introduced by Ng et al. [3], provides a theoretical foundation for designing potential-based reward functions that guarantee policy invariance. However, manually designing such functions remains a significant challenge. Our work automates the generation of shaping signals, though we do not formally enforce the potential-based constraint, prioritizing practical performance and flexibility.

In the context of LLMs for RL, many works have focused on using LLMs as high-level planners or to generate executable code. For instance, some approaches prompt LLMs to write the reward function code directly within a script [8], while others generate entire policy programs . Our method differs by proposing a decoupled, structured representation (the reward patch), which enhances modularity and interpretability. It allows a human engineer to easily verify and edit the LLM's output without touching the agent's source code, fostering a more robust human-in-the-loop workflow.

V. Conclusion

This paper presented a novel framework for accelerating Multi-Agent Reinforcement Learning by using Large Language Models to generate structured reward patches. Our experiments demonstrated that by fine-tuning a pre-trained policy with an LLM-generated patch, we can achieve a significant 3.6-fold average improvement in sample efficiency on a canonical coordination task, with best-case results showing up to a 46-fold acceleration. A key finding from our ablation study is that this success stems from the powerful synergy between the general skills learned during pre-training and the dense, task-specific guidance provided by the patch.

The proposed method offers a modular, interpretable, and highly efficient paradigm for reward engineering. Future work will involve applying this framework to more complex, 3D environments and exploring methods for enabling the LLM to iteratively refine the reward patch based on agent performance feedback.

Acknowledgment

I would like to express my heartfelt gratitude to my family and friends for their unwavering support and encouragement throughout the research and writing process. Their invaluable emotional and intellectual support has been instrumental in the completion of this work.

References

[1] R. Lowe, Y. Wu, A. Tamar, J. Harb, P. Abbeel, and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments," 2017. [Online]. Available: https://arxiv.org/abs/1706.02275

[2] J. Foerster, G. Farquhar, T. Afouras, N. Nardelli, and S. Whiteson, "Counterfactual multi-agent policy gradients," 2017. [Online]. Available: https://arxiv.org/abs/1705.08926

[3] A. Y. Ng, D. Harada, and S. Russell, "Policy invariance under reward transformations: Theory and application to reward shaping," in *Machine learning: Sixteenth international conference on machine learning(ICML'99), June 27-30, 1999, Bled, Slovenia*, 1999.

[4] D. Silver, S. Singh *et al.*, “Reward is enough,” *Artificial Intelligence*, 2021.

[5] OpenAI, “Gpt - 4 technical report,” *arXiv preprint arXiv:2303.08774*, 2023. [Online]. Available: https://arxiv.org/abs/2303.08774

[6] X. Wang and et al., “A survey on large language model based autonomous agents,” *Frontiers of Computer Science*, 2023. [Online]. Available: http://dx.doi.org/10.1007/s11704-024-40231-1

[7] C. Yu, A. Velu, E. Vinitsky, J. Gao, Y. Wang, A. Bayen, and Y. Wu, “The surprising effectiveness of ppo in cooperative, multi-agent games,” 2021. [Online]. Available: https://arxiv.org/abs/2103.01955

[8] W. Yu, N. Gileadi, C. Fu, S. Kirmani, K.-H. Lee, M. G. Arenas, H.-T. L. Chiang, T. Erez, L. Hasenclever, J. Humplik, B. Ichter, T. Xiao, P. Xu, A. Zeng, T. Zhang, N. Heess, D. Sadigh, J. Tan, Y. Tassa, and F. Xia, “Language to rewards for robotic skill synthesis,” 2023. [Online]. Available: https://arxiv.org/abs/2306.08647