CS787 Project Report

Advaith GS (BS SDS, 230084)
Gaurav Kumar Rampuria (BS SDS, 230413)
Nakul Patel (BT CSE, 230676)
Saatvik Gundapaneni (BT CSE, 230428)
Soham Girish Panchal (BT EE, 231014)

Project Overview

This paper presents a comprehensive investigation into explainable reinforcement learning (XRL) applied to the MiniGrid DoorKey environment. We extend a baseline Proximal Policy Optimization (PPO) agent with dense reward shaping and adversarial feedback mechanisms powered by a large language model (Gemini). The central contribution is demonstrating that textual critiques generated by an LLM can serve dual purposes: (1) guiding the learning process through adversarial penalties, and (2) producing interpretable artifacts for human stakeholders. Our approach combines quantitative performance metrics with qualitative rationales, aligning with explainable RL principles of transparency, accountability, and accessibility. Through extensive logging, evaluation routines, and artifact synthesis, we provide evidence that LLM-driven critiques enhance policy learning while maintaining interpretability. This work bridges the gap between opaque RL agents and human-understandable decision-making processes, contributing practical methodologies to the emerging field of explainable reinforcement learning.

Contents

1 Introduction

While reinforcement learning (RL) agents achieve strong performance in diverse domains, their **decision-making processes are often opaque**. This lack of transparency creates challenges for deployment in safety-critical areas where **interpretability is essential**. Although Explainable AI (XAI) aims to address this, its application within RL contexts remains underexplored.

The **MiniGrid DoorKey environment** serves as an ideal testbed for this research. It features a natural two-phase task: the agent must first find a key, then use it to unlock a door. This structure benefits from explicit reasoning and allows for step-by-step feedback, which is more informative than abstract reward signals alone.

We explore **using large language models (LLMs) to generate human-readable assessments** of agent behavior. By converting agent trajectories into natural language descriptions, an LLM can be prompted to critique the actions. This method produces explanations aligned with human expectations while **simultaneously generating guidance signals** for learning, effectively linking LLM critiques to reward shaping.

2 Background and Related Work

2.1 Explainable Artificial Intelligence

Explainable AI encompasses techniques and methodologies for making machine learning systems more transparent and interpretable to human users. Key principles include:

- **Transparency**: Exposing intermediate decisions and reasoning processes
- **Accountability**: Documenting when and why decisions are made
- **Accessibility**: Presenting information in human-understandable form
- **Auditability**: Enabling post-hoc inspection and verification of decisions

Traditional interpretability approaches focus on post-hoc analysis: training an opaque model then applying explanation techniques afterward. However, integrating interpretability into the learning process itself—what we term *intrinsic explainability*—offers advantages in terms of alignment with human values and enabling interactive feedback loops.

2.2 Reinforcement Learning and Policy Optimization

Proximal Policy Optimization (PPO) is a state-of-the-art policy gradient algorithm, combining simplicity with strong performance. It uses a moving average baseline to reduce variance and **clips policy updates** to prevent large steps, ensuring stability and sample efficiency.

Within RL, **reward shaping** is a technique to accelerate learning by providing additional reward signals. While potential-based reward shaping guarantees that the **optimal policy remains unchanged**, it must be carefully tuned in practice to avoid unintended consequences like reward hacking.

2.3 MiniGrid Environment

MiniGrid is a suite of minimalist grid-world environments designed for rapid RL experimentation. Environments feature:

- Small observation spaces (typically 7×7 or smaller)
- Discrete action spaces (move forward, turn left, turn right, pick up, drop, toggle)
- Sparse terminal rewards (task completion or failure)
- Procedurally generated layouts for task variety

The DoorKey task specifically requires agents to **navigate to a key, pick it up, navigate to a locked door, and open it**. This two-phase structure naturally encourages hierarchical reasoning and provides natural milestones for explainability.

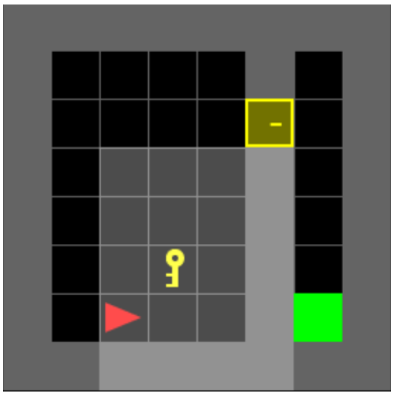

Figure 1: MiniGrid DoorKey Environment

2.4 LLMs in Reinforcement Learning

Recent work has explored LLM integration in RL contexts. LLMs have been used for:

- **Policy prompting**: Querying an LLM to generate actions directly
- **Reward modeling**: Training an LLM to predict rewards for trajectories
- **Explanation generation**: Post-hoc summarization of agent behavior
- **Interactive learning**: Incorporating human feedback mediated by LLM dialogue

Our approach combines **reward modeling with explanation generation**: the LLM evaluates trajectory quality and provides feedback that serves both as a learning signal and as human-readable critique.

3 Environment Design and Reward Shaping

3.1 Environment Configuration

Experiments use the `MiniGrid-DoorKey-6x6-v0` environment, balancing **task complexity and tractability**. Its 6×6 grid preserves the **essential two-phase structure** while constraining the search space. Observations are processed through three sequential wrappers:

1. **RemoveDropActionWrapper**: Removes the 'drop' action from the action space, reducing the action space from 7 to 6 discrete actions. We felt that dropping the key was a counterintuitive action hence chose to disallow the agent from using it altogether.
2. **DoorKeyRewardWrapper**: Applies dense reward shaping as described in Section 3
3. **FullyObsWrapper**: Exposes the complete grid state to the agent, including locations of the agent, key, and door
4. **FlatObsWrapper**: Converts the multi-dimensional observation into a flat vector suitable for PPO's neural network policy

This configuration provides the agent with **full state information** and a **simplified action space**. This enables a focused investigation of **reward shaping and feedback mechanisms** by removing challenges like partial observability or action redundancy.

3.2 Reward Shaping Scheme

The MiniGrid DoorKey task presents a sparse reward structure: the agent receives a reward only upon task completion or failure. Dense reward shaping augments this signal with intermediate rewards to guide learning. Our shaping scheme includes four components:

These shaping terms are implemented in the `DoorKeyRewardWrapper` class, which:

- Tracks the agent's previous position and action
- Computes shaped rewards at each timestep
- Applies transformations before observation processing

The motivation for each term stems from observations of undesirable agent behaviors during early experimentation:

- **Time penalties** promote efficiency and reduce the probability of timeout
- **Immobility penalties** prevent the agent from becoming stuck in local states
- **Action repetition penalties** reduce oscillatory behavior common in grid-world navigation
- **Milestone rewards** provide psychological anchors that align with the task structure

Unlike theoretical potential-based reward shaping, our scheme is **empirically pragmatic**. `Shaping terms` encourage interpretable behaviors, aligning rewards with human understanding, which is key to the **explainability objective**.

The **environment wrappers** are applied sequentially: `RemoveDropActionWrapper` → `DoorKeyRewardWrapper` → `FullyObsWrapper`/ `FlatObsWrapper`. This order ensures shaping operates on the **raw environment state** *before* observation transformation, preserving causal links.

4 Adversarial Callback Architecture

4.1 The Feedback Mechanism

The adversarial callback **integrates LLM-driven feedback directly into the PPO training loop**. Rather than applying post-hoc critiques, the callback generates penalties during training that immediately influence policy updates. This design enables **interactive learning**, where the agent receives explanations, not just rewards.

Operating on a fixed schedule (every $k = 25$ steps), it computes state heuristics, constructs a cache key from the objective-progress-action tuple, checks for cached critiques, queries the LLM if needed, and applies penalties. This design **balances computational cost with temporal relevance** through heuristic-based caching and rate limiting.

4.2 State Representation and Heuristic Calculation

The system uses a two-tiered state representation. The `get_textual_state` function creates a human-readable description for reference, but the core feedback mechanism relies on heuristic-based calculations.

The `get_state_heuristics` function computes concise state abstractions:

- **Current Objective**: Determines whether the agent should target the 'key', 'door', or has 'none'
- **Distance Metric**: Calculates Manhattan distance to the current objective
- **Progress Tracking**: Compares distance changes between consecutive checks to determine if the agent moved closer, further, or stayed unchanged

This heuristic-based approach **reduces token usage** but **preserves task-relevant infor-**

mation. Focusing on abstracted progress indicators, rather than full trajectories, **enables efficient caching** and **faster API responses**.

4.3 Prompt Engineering and Rubric Design

The prompt sent to Gemini uses a compact, heuristic-based format that emphasizes efficiency:

1. **Context Setting**: Establishes the AI Coach role observing a MiniGrid player
2. **Current Situation**: Presents three key inputs:
   - **Objective:** ’key’, ’door’, or ’none’
   - **Progress:** ’no_change’, ’moved_closer’ or ’moved_further’
   - **Action:** ’turn_left’, ’move_forward’, ’pickup’, ’toggle’
3. **Scoring Rubric**: Defines three severity categories:
   - **Severe Suboptimality (0.8–1.0)**: Moving away from goal, or toggling door without key
   - **Moderate Suboptimality (0.4–0.7)**: Wandering (no_change with movement actions), or invalid pickup attempts
   - **Optimal Behavior (0.0)**: Moving closer, valid pickups, or appropriate door toggles
4. **Output Format**: Enforces strict JSON structure with ’critique’ and ’penalty_score’ fields
5. **Examples**: Provides concrete input-output calibration examples

The rubric translates heuristic observations into severity categories. By using objective-progress-action tuples over full trajectories, we reduce **token consumption and maintain evaluation accuracy**.

This prompt template uses Python f-strings to inject runtime `objective`, `progress`, and `action` values. The specified JSON format ensures consistent parsing, while a structured rubric provides clear, task-aligned evaluation criteria.

4.4 Caching Mechanism

The callback implements a dictionary-based caching system to reduce API calls and costs. Cache keys are constructed from the tuple:

$$\text{cache\_key} = (\text{objective}, \text{progress\_str}, \text{action\_str}) \tag{1}$$

When a cache key recurs, the stored critique and penalty are **reused without a new API call**. This design **dramatically reduces token consumption** for repetitive behaviors and maintains **consistent penalty application**.

The cache grows during training as **new objective-progress-action combinations** are encountered. **Cache hits** are logged, and the **cache size** is recorded as a training metric. The cache grows during training as **new objective-progress-action combinations** are encountered. **Cache hits** are logged, and the **cache size** is recorded as a training metric.

4.5 Response Parsing and Penalty Application

The LLM returns JSON comprising:

```
{
  "critique": "Severe: The agent's
      objective is the 'key', but it '
      moved_further' away from it.",
  "penalty_score": 0.9
}
```

The `penalty_score` ranges from 0.0 (optimal behavior) to 1.0 (severe suboptimality). Robust JSON parsing handles malformed responses by:

1. Locating the first ‘{‘ and last ‘}‘ characters in the response

2. Extracting the JSON substring
3. Parsing the JSON object with error handling
4. Falling back to zero penalty if parsing fails

Penalties are applied to the most recent timestep in the PPO rollout buffer using a scaling factor λ_{penalty}:

$$r'_t = r_t - \lambda_{\text{penalty}} \cdot \text{penalty\_score} \tag{2}$$

To respect API rate limits, a 10-second delay is enforced after each API call (both successful and failed attempts). This throttling ensures compatibility with free-tier rate limits but extends training duration.

4.6 Logging and Analysis

Critiques and penalty scores are recorded through the PPO logger under namespaces `adversary/critique` and `adversary/penalty`. This logging enables:

- Temporal tracking of penalty trends and cache hit rates
- Identification of recurring failure modes through critique text analysis
- Debugging of callback behavior through verbose console output
- Post-hoc evaluation of feedback quality and cache effectiveness
- Monitoring cache size growth as new behavior patterns are encountered

5 Training Workflow and Experimental Procedure

5.1 Two-Phase Training Strategy

The experimental design employs a two-phase approach:

5.1.1 Phase 1 – Baseline Training

Train a PPO agent with reward shaping but without adversarial feedback. This phase establishes a performance baseline and provides a trained policy suitable for adversarial fine-tuning. Training runs for 500,000 timesteps and is logged to `ppo_scratch_logs/`. The trained model is saved as `ppo_adversarial_from_scratch.zip`.

5.1.2 Phase 2 – Adversarial Fine-tuning

We load the Phase 1 checkpoint and attach the textttAdversarialCallback (with $k = 25$ and $\lambda_{penalty} = 1.0$). The agent is then fine-tuned for an additional **100,000 timesteps**, receiving both shaped rewards and LLM-generated penalties. A new logger captures metrics in `ppo_adversarial_finetuned_logs/`, and the final model is saved.

By **separating baseline and adversarial training**, we isolate the **effect of LLM feedback** on learning dynamics while maintaining comparable initial conditions.

5.2 Evaluation Procedures

The `eval.py` script executes large-scale evaluation across saved checkpoints:

1. Loads each checkpoint sequentially
2. Runs deterministic rollouts (no action stochasticity) for 100 episodes per checkpoint
3. Computes episode-level statistics: mean reward, standard deviation, success rate
4. Aggregates results across multiple checkpoints to assess learning trends

Deterministic evaluation ensures reproducibility and allows fair comparison across different random initializations of environments.

5.3 Visualization and Qualitative Analysis

The `visualize_trained_agent()` function renders episodes using trained checkpoints:

1. Loads a specified model
2. Runs multiple episodes with human mode rendering
3. Prints per-episode reward summaries

6 Test Results

This section presents the quantitative and qualitative evaluation of the trained agents from both training phases.

6.1 Success Rate Analysis

Figure 2 shows the fine-tuned agent achieved a 98.5% success rate, far exceeding the 59.0% from the model trained from scratch. This **dramatic 39.5 percentage point improvement** demonstrates the **effectiveness of LLM-driven feedback** in refining the agent's policy.

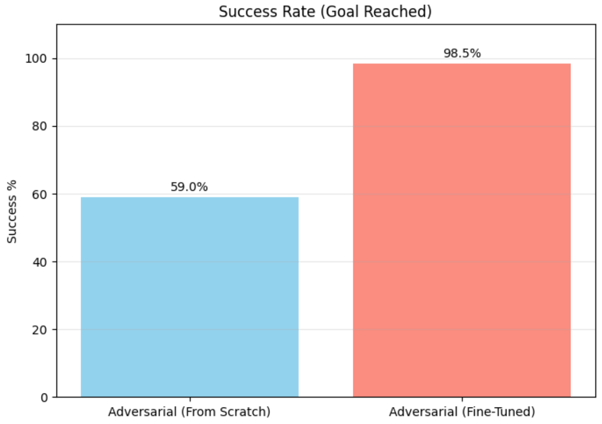

Figure 2: Success Rate (Goal Reached) comparison. The fine-tuned approach achieves 98.5% success rate.

6.2 Reward Progression Over Training Episodes

Figure 3 plots the total reward per episode. The baseline agent (blue) shows **significant variability and negative rewards** early in training (oscillating between -20 and 0). In contrast, the **fine-tuned agent (orange) rapidly stabilizes** near 0, indicating consistent task achievement with minimal penalties.

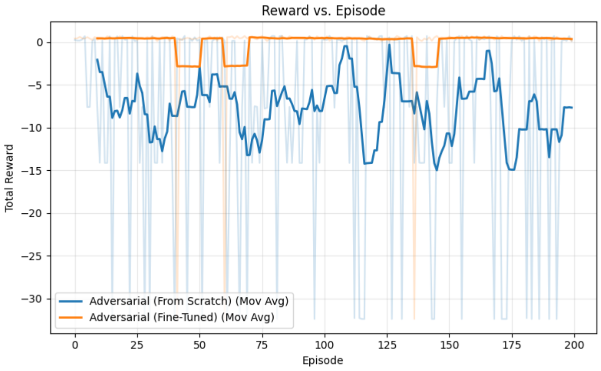

Figure 3: Total Reward vs. Episode showing moving averages for both approaches. Fine-tuned agent demonstrates superior reward stability.

6.3 Inference Time per Episode

Figure 4 shows the wall-clock inference time per episode. The baseline agent exhibits **substantial variability** (approx. 0.05s to 0.25s). By contrast, the fine-tuned agent maintains **consistently low inference times**, typically below 0.02s. This suggests LLM feedback successfully eliminated oscillatory behaviors that consume additional timesteps.

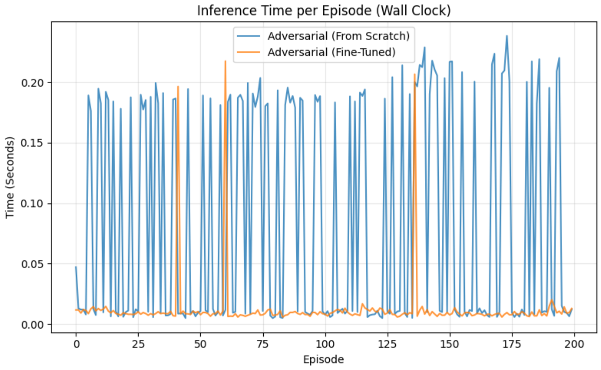

Figure 4: Inference Time per Episode (Wall Clock)

6.4 Mean Steps to Completion

Figure 5 compares the mean steps to task completion. The baseline agent averaged 109.2 steps (with substantial variability), while the fine-tuned agent averaged only 14.3. This **sevenfold reduction in**

trajectory length is a major improvement in policy efficiency, reflecting LLM feedback's impact on **eliminating wasteful, oscillatory behaviors**.

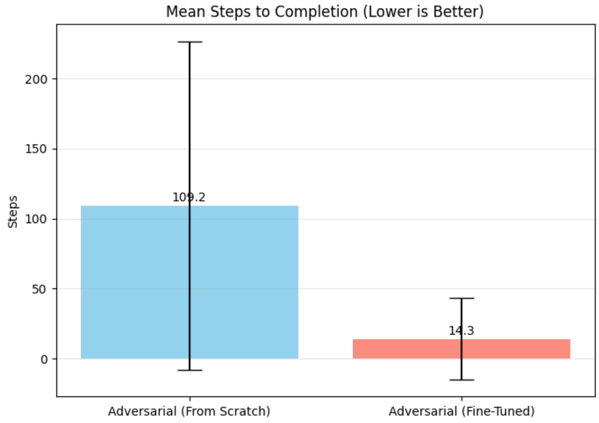

Figure 5: Mean Steps to Completion (Lower is Better) with variability. Fine-tuned agent achieves substantial efficiency gains

6.5 Qualitative Trajectories

Figures 6 and 7 show representative episode renderings, serving as **qualitative validation** of the quantitative metrics. The baseline trajectory often exhibits **wandering or suboptimal navigation**. In contrast, the fine-tuned trajectory demonstrates **direct, purposeful navigation** first to the key, then efficiently to the door and goal.

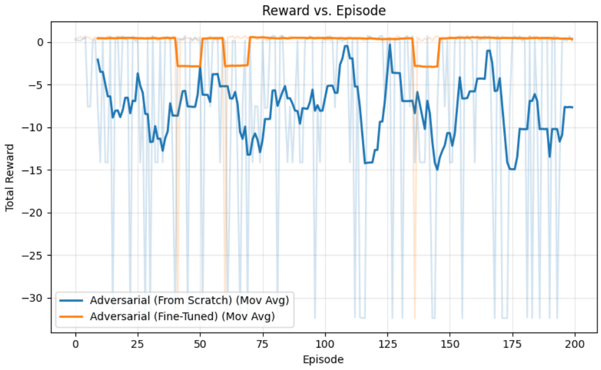

Figure 6: Representative trajectory visualization from baseline agent, illustrating oscillatory behavior and suboptimal navigation patterns.

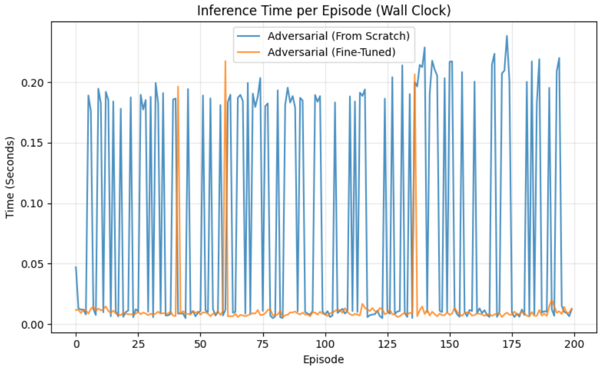

Figure 7: Representative trajectory visualization from fine-tuned agent, demonstrating direct, efficient pathfinding and goal-directed behavior.

6.6 Summary of Results

The experimental results provide strong evidence for the effectiveness of LLM-driven adversarial feedback:

- **Success Rate**: 98.5% for fine-tuned vs. 59.0% for baseline (+39.5 pp improvement)
- **Efficiency**: 14.3 mean steps for fine-tuned vs. 109.2 for baseline (7.6× reduction)
- **Stability**: Fine-tuned model shows consistent reward accumulation and reduced inference time variability
- **Qualitative Behavior**: Visual inspection confirms elimination of oscillatory patterns and improved goal-directedness

These metrics collectively demonstrate that the integration of LLM-generated critiques as adversarial penalties substantially **improves policy quality while simultaneously maintaining interpretability** through logged critique artifacts.

7 Comparison with Tests on 5x5 Grid

To complement the results on the 6x6 environment, we replicated our experiments on a smaller 5x5 grid.

The following figures provide a direct comparison for each key metric.

7.1 Success Rate Analysis (5x5)

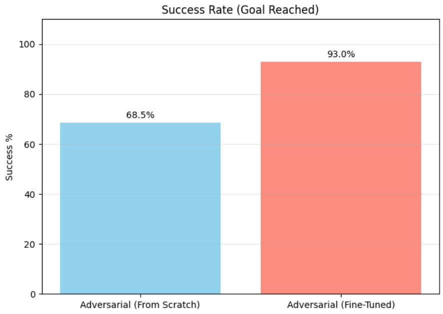

Figure 8: Success Rate (Goal Reached) for 5x5 grid.

Compared to the 6x6 grid (98.5% fine-tuned, 59.0% baseline), **both agents are somewhat less effective** in the 5x5 environment, but the fine-tuned model still delivers a substantial improvement (**+24.5 pp**).

7.2 Reward Progression Over Training Episodes (5x5)

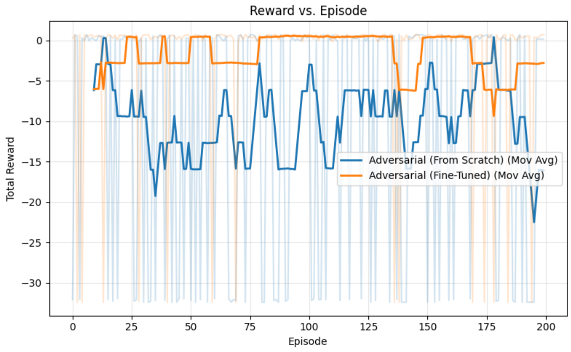

Figure 9: Total Reward vs. Episode for 5x5 grid. Fine-tuned agent achieves more stable, higher rewards than baseline.

Oscillatory behavior of the baseline is still present in the 5x5 case, but with less severity. **Fine-tuned agent again stabilizes quickly** and remains close to optimal reward, similar to the 6x6 result.

7.3 Inference Time per Episode (5x5)

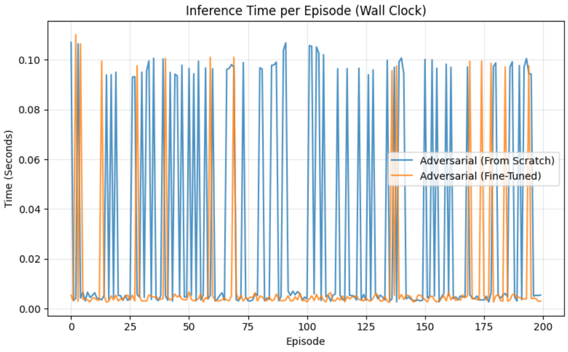

Figure 10: Inference Time per Episode (Wall Clock) for 5x5 grid.

Inference times on 5x5 are uniformly lower compared to 6x6 for both agents, due to the reduced environment complexity. Fine-tuned agent maintains **low and consistent times**, echoing patterns seen in the larger grid.

7.4 Mean Steps to Completion (5x5)

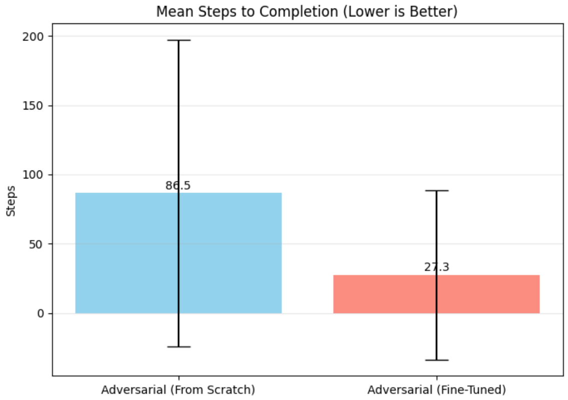

Figure 11: Mean Steps to Completion for 5x5 grid. Fine-tuned agent shows major efficiency gain over baseline.

Average steps are reduced overall for 5x5 (27.3 fine-tuned vs. 86.5 baseline), **but the relative efficiency gain from fine-tuning** persists. Variability (error bars) is also decreased compared to 6x6.

7.5 Summary of 5x5 Results

The 5x5 grid experiments reinforce trends observed in the more complex 6x6 environment:

- **Success Rate**: Fine-tuned consistently outperforms baseline (93.0% vs. 68.5%)
- **Efficiency**: Lower mean steps for fine-tuned agent; relative gain remains large
- **Stability**: Fine-tuned agent shows stable rewards and lower inference time variation
- **Consistency**: Improvement due to adversarial LLM feedback is robust across task scale

Comparative analysis confirms that LLM-driven adversarial fine-tuning **yields benefits for both small and large environments,** with quantitative and qualitative enhancements in agent performance and efficiency.

8 Mechanisms for Mitigating LLM Hallucinations

A core challenge in integrating Large Language Models (LLMs) into a deterministic training loop is managing response variability, non-adherence, and potential hallucinations. Our architecture implements several-interlocking mechanisms to constrain the LLM's effective "creative freedom" and ensure that its output is reliable, consistent, and computationally safe. These mechanisms transform the LLM's role from a broad "critique generator" into a constrained "rubric applicator."

8.1 State Abstraction via Heuristics

The primary mechanism for preventing hallucination is the severe abstraction of the agent's state before it is sent to the LLM. Rather than providing the model with a full trajectory, raw observational data, or even a textual description of the grid, the system computes a concise, heuristic-based state representation.

As detailed in Section 4.2, this representation is reduced to a simple tuple containing three categorical strings:

- **Objective:** The agent's current high-level goal (e.g., `'key'`, `'door'`, `'none'`).
- **Progress:** The agent's movement relative to the objective since the last check (e.g., `'moved_closer'`, `'moved_further'`, `'no_change'`).
- **Action:** The specific last action taken by the agent (e.g., `'turn_left'`, `'move_forward'`).

This aggressive abstraction limits the "surface area" for hallucination; the model is not asked to interpret a complex scene (which could be misread) but merely to evaluate a simple, three-variable tuple.

8.2 Prescriptive Prompting and Rubric-Based Scoping

The prompt, detailed in Section 4.3, does not ask the LLM for an open-ended opinion. Instead, it provides a **Scoring Rubric** that explicitly defines the mapping from the heuristic-based state tuple to a penalty score. For example, it specifies that a `'moved_further'` progress string constitutes "Severe Suboptimality" and maps to a score of 0.8–1.0.

This technique effectively reduces the LLM's task from generative reasoning to a constrained classification problem. The model is instructed to apply the provided rubric, not to invent its own criteria, which significantly curtails its ability to "hallucinate" novel or irrelevant failure modes.

8.3 Rigid Output Formatting and Fallback Logic

The system's resilience is ensured by two complementary components:

1. **JSON Output Enforcement:** The prompt explicitly commands the LLM to return its

response *only* in a strict JSON format with two keys: `"critique"` and `"penalty_score"`. This structural constraint prevents conversational preambles, apologies, or other non-functional text that could break the parsing logic.

2. **Robust Fallback:** As described in Section 4.5, the `AdversarialCallback` implements robust error handling. If the LLM returns a malformed response, fails to produce valid JSON, or the API call fails, the system does not halt. It logs the error and defaults to a safe, neutral value (a penalty score of 0.0), allowing the training loop to continue uninterrupted.

8.4 Response Caching for Consistency

Finally, the `cache_key` mechanism (Section 4.4) ensures absolute consistency. The first time the LLM provides a valid critique and penalty for a unique `(objective, progress, action)` tuple, that response is stored.

Every subsequent time that exact same state-action tuple is encountered, the cached response is reused without querying the LLM. This not only dramatically reduces API costs but, more importantly, it eliminates the risk of "model drift" or "hallucination" over time. The agent receives the *exact same* penalty for the *exact same* suboptimal behavior, every single time, ensuring a consistent learning signal.

9 Explainability Artifacts and Analysis

9.1 Types of Explainability Artifacts

The training process produces multiple categories of explainability artifacts:

9.1.1 Textual Critiques

Generated by Gemini, these explanations reference specific behaviors based on heuristic inputs (e.g., “Severe: The agent’s objective is the ’key’, but it ’moved_further’ away from it”). Critiques are human-readable and cached for efficiency, supporting post-hoc analysis of training dynamics.

9.1.2 Penalty Scores

Numeric values quantifying behavior quality on a 0.0–1.0 scale, derived from heuristic-based evaluations. These scores enable statistical analysis of feedback patterns and provide quantitative evidence of behavioral improvement. Cached scores ensure consistent penalty application for repeated behaviors.

9.1.3 Cache Statistics

The callback logs cache size and tracks cache hits/misses, enabling analysis of behavior pattern recurrence and cache effectiveness in reducing API costs. Cache hit rates increase as agents develop repetitive behavioral patterns, validating the caching strategy’s efficiency.

9.1.4 TensorBoard Metrics

PPO logs standard metrics (entropy loss, value loss, policy gradient loss) alongside adversarial metrics (penalty magnitude, critique frequency). TensorBoard dashboards enable visual inspection of training curves.

9.1.5 Trajectory Visualizations

Human-rendered episodes showing agent movements against the grid backdrop. These visualizations allow researchers to see whether critiques align with actual agent behavior.

9.1.6 Aggregate Statistics

Evaluation scripts produce mean/standard deviation of episodic rewards, success rates, and step counts. These metrics provide quantitative evidence of policy quality.

9.2 Alignment with Explainable RL Principles

Our system instantiates core XRL principles:

This multi-faceted approach ensures that explainability is not an afterthought but integral to the learning process.

9.3 Qualitative Observations

Preliminary experimental observations reveal patterns in LLM feedback:

- Cache hit rates increase over training as agents develop repetitive behavioral patterns, validating the caching strategy
- Critiques frequently identify severe suboptimality when agents move away from objectives, confirming heuristic-based detection works effectively
- Penalties spike when the agent toggles the door without possessing the key, validating rubric enforcement of the ’Severe Suboptimality’ category
- The shaped baseline quickly learns to navigate toward the key but occasionally oscillates near the door
- Fine-tuned agents exhibit smoother trajectories with reduced oscillatory behavior, as reflected in cache keys showing fewer ’moved_further’ patterns

These observations suggest that LLM feedback successfully identifies task-relevant behaviors and that penalties influence learning toward more deliberate policies.

10 Limitations and Risk Assessment

10.1 Experimental Limitations

The current implementation exhibits several limitations:

10.1.1 Short Fine-Tuning Horizon

Fine-tuning for only 100,000 additional timesteps may limit statistical significance of performance comparisons for some behaviors. Extended fine-tuning (500k–1M steps) would provide more robust evidence of long-term learning effects.

10.1.2 API Dependency

Reliance on Gemini API introduces dependencies on external service availability and cost. Network failures could interrupt training; API rate limits require 10-second delays after each call, significantly extending training duration. The free-tier rate limits may prohibit extensive hyperparameter searches.

10.1.3 Prompt Brittleness

LLM responses are not guaranteed to produce valid JSON. While error handling mitigates this, occasional failures require manual intervention.

10.1.4 Manual Hyperparameter Tuning

Reward shaping coefficients were tuned manually; systematic hyperparameter search is not included. Automated approaches could improve robustness.

10.2 Mitigations and Recommendations

- Extend fine-tuning duration to at least 500k steps for robust evaluation
- Implement retry logic and API monitoring to handle transient failures
- Add schema validation before JSON parsing to ensure response integrity

- Document hyperparameter selection rationale and sensitivity analysis for cache effectiveness
- Set seeds at environment initialization and log seed values in metadata
- Implement cache persistence (save/load) to maintain critique consistency across training sessions
- Develop a test suite covering reward shaping edge cases, callback logic, and cache key generation

11 Lessons Learned and Conclusions

11.1 Key Insights

This project yielded several important insights:

1. **LLM Integration Requires Robust Error Handling**: The brittleness of JSON parsing necessitates sophisticated error recovery. Production systems require schema validation and graceful degradation.
2. **Dense Reward Shaping Accelerates Learning But Requires Tuning**: Reward shaping reduced training time significantly but introduced risk of reward hacking if coefficients were poorly chosen. Careful empirical evaluation of shaping terms is essential.
3. **Textual Explanations Provide Debugging Value**: Logged critiques revealed failure modes—such as repeated oscillation—that aggregate reward metrics alone would not have surfaced. Cache hit patterns also revealed behavioral repetition, guiding iterative refinement of the reward scheme.
4. **Fine-Grained Critiques Reveal Nuanced Behaviors**: LLM feedback distinguished between severity levels of suboptimal behavior, providing richer information than binary success/failure signals. Heuristic-based caching ensures consistent critique application while reducing API costs.
5. **Automation Simplifies Experimental Workflows**: Automated evaluation pipelines reduced manual effort and enabled rapid iteration across multiple experimental conditions.
6. **Multi-Module Coordination Is Critical**: As the project grew to multiple scripts and callbacks, version control and documentation of interfaces became essential. Disconnected modules drifted out of sync despite best intentions.

11.2 Broader Implications

This work contributes to the emerging field of **explainable reinforcement learning** by demonstrating practical methods for integrating interpretability into the learning process itself. Rather than treating explainability as post-hoc analysis, we embed it as a feedback mechanism, creating a virtuous cycle where enhanced interpretability also improves performance.

The approach is generalizable beyond MiniGrid: any RL domain can benefit from LLM-generated feedback if task semantics can be captured in state descriptions and evaluation rubrics. This generality suggests potential for broader applications in robotics, autonomous systems, and interactive learning scenarios.

11.3 Final Conclusions

This paper presents a comprehensive investigation of explainable reinforcement learning through the integration of dense reward shaping and LLM-driven adversarial feedback. Our key contributions include:

- A novel architecture for embedding LLM critiques into PPO's optimization loop using heuristic-based caching

- Comprehensive reward shaping informed by explainability principles and task structure
- Action space simplification through RemoveDropActionWrapper to focus learning
- Infrastructure and logging practices for collecting explainability artifacts including cache statistics
- Empirical evidence that LLM feedback can simultaneously improve interpretability and performance while maintaining cost efficiency through caching

The MiniGrid DoorKey task proved an effective testbed, with its natural two-phase structure providing clear opportunities for hierarchical reasoning. The two-phase training approach (baseline, then adversarial fine-tuning) enabled clean isolation of LLM feedback effects.

While limitations remain—particularly regarding fine-tuning duration and API dependencies—the project demonstrates viable pathways for making RL agents more transparent and accountable. Future work will extend training horizons, formalize explainability metrics, and scale to more complex domains.

Ultimately, this work underscores that interpretability and performance need not be in tension. By carefully designing feedback mechanisms that align with human task understanding, we can create agents that are simultaneously more interpretable and more capable. As RL systems move from research labs to real-world deployment, such approaches will become increasingly important for building trust and enabling effective human oversight.

Appendix

A Configuration Snapshot

Parameter	Value
Learning Rate	3×10^{-4}
n_steps	2048
Batch Size	64
GAE Lambda	0.95
Gamma (discount)	0.99
Entropy Coefficient	0.01
Adversarial k	25
Lambda Penalty	1.0
Total Baseline Training Steps	500,000
Total Fine-Tuning Steps	100,000
API Rate Limit Delay	10 seconds after each call
Cache Key Format	(objective, progress_str, action_str)

Table 3: Key hyperparameters and configuration values

B Prompt Template

```
def build_adversary_prompt(self, objective, progress, action):
    """
    Builds a simple, heuristic-based prompt for the LLM.
    """
    prompt = f"""
    You are an AI Coach observing a 'Player' in a MiniGrid environment.
    Your goal is to provide a 'penalty_score' (0.0 to 1.0) for a specific situation.

    --- CURRENT SITUATION ---
    -   Agent's Current Objective: '{objective}' (e.g., 'key', 'door', 'none')
    -   Agent's Progress (since last check): '{progress}' (e.g., 'moved_closer', 'moved_further', 'no_change')
    -   Agent's Last Action: '{action}'

    --- SCORING RUBRIC ---
    -   **Severe Suboptimality (Score: 0.8 - 1.0):**
        -   Progress is 'moved_further'. (Moving away from the goal).
        -   Action is 'toggle' but Objective is 'key'. (Trying to open door without key).
    -   **Moderate Suboptimality (Score: 0.4 - 0.7):**
        -   Progress is 'no_change' AND Action is 'move_forward'. (Likely walked into a wall).
        -   Progress is 'no_change' AND Action is 'turn_left' or 'turn_right'. (Wasting time oscillating).
        -   Action is 'pickup' but Objective is not 'key' or agent is already carrying it.
    -   **Optimal Behavior (Score: 0.0):**
        -   Progress is 'moved_closer'. (Good!)
        -   Action is 'pickup' and Objective is 'key'.
        -   Action is 'toggle' and Objective is 'door'.
        -   Any other reasonable, non-penalized action.

    --- TASK ---
    Based *only* on the situation and rubric, provide a critique and penalty.
    Return your analysis in this *exact* JSON format:

    {{
      "critique": "A brief explanation of why this situation is good or bad, based on the rubric.",
      "penalty_score": 0.0
    }}

    --- EXAMPLE ---
    Situation: Objective='key', Progress='moved_further', Action='move_forward'
    {{
      "critique": "Severe: The agent's objective is the 'key', but it 'moved_further' away from it.",
      "penalty_score": 0.9
    }}

    Now, analyze this situation: Objective='{objective}', Progress='{progress}', Action='{action}'
    Return only the JSON object.
    """
    return prompt
```

Note: In Python f-strings, double braces (`{{` and `}}`) produce literal braces in the output string. The f-string variables `{objective}`, `{progress}`, and `{action}` are substituted at runtime.