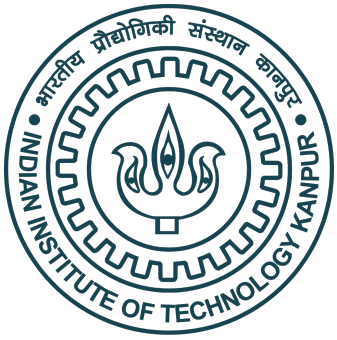

CS787 Project Report

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

Reinforcement learning (RL) agents achieve remarkable performance across diverse domains, from game-playing to robotics, yet their decision-making processes remain largely opaque to human observers. The agent optimizes cumulative reward without exposing the reasoning behind individual actions, creating challenges for deployment in safety-critical domains where interpretability is paramount. Explainable AI (XAI) addresses this opacity by producing explanations understandable to users, but explainability remains underexplored in RL contexts.

The MiniGrid DoorKey environment offers an ideal testbed for explainable RL research. This environment presents a two-phase task structure: agents must first obtain a key, then use it to unlock a door and reach the goal. This natural decomposition benefits from explicit reasoning and step-by-step feedback. Unlike abstract reward signals alone, human-readable explanations can guide agents toward interpretable policies and help researchers debug suboptimal behavior.

Recent advances in large language models (LLMs) have created new opportunities for generating human-readable assessments of agent behavior. By converting agent trajectories into natural language descriptions and prompting an LLM to critique the actions, we can produce explanations that align with human expectations while simultaneously generating guidance signals for learning. This paper explores this intersection of reward shaping, LLM critique generation, and transparent logging practices.

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 that combines simplicity with strong empirical performance. PPO maintains a moving average baseline to reduce gradient variance and clips policy updates to prevent excessively large

steps. These properties make PPO stable and sample-efficient for continuous and discrete control tasks.

Within the RL literature, reward shaping has long been recognized as a technique to accelerate learning by providing agents with additional reward signals beyond those provided by the environment. Potential-based reward shaping, introduced by Ng *et al.*, provides theoretical guarantees that shaping does not change the optimal policy. However, in practice, reward shaping must be carefully tuned to avoid unintended consequences such as 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.

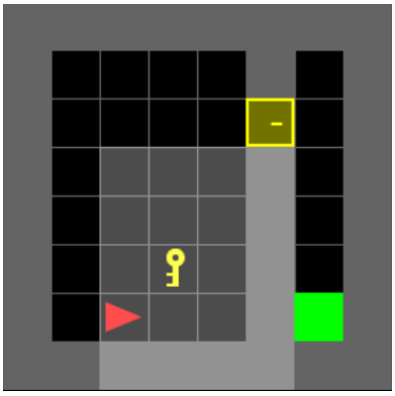

Figure 1: MiniGrid DoorKey Environment

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

The main experiments use the `MiniGrid-DoorKey-6x6-v0` environment, providing a balance between task

complexity and computational tractability. The 6×6 grid constrains the search space to a manageable size while preserving the essential two-phase structure of the task. 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. This wrapper remaps agent actions (0-5) to environment actions (0-6, skipping action 4 'drop'). This simplification focuses the agent on navigation and interaction tasks without the complexity of inventory management.

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, enabling focused investigation of reward shaping and feedback mechanisms without the additional challenge of partial observability or action space 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

While potential-based reward shaping provides theoretical guarantees under certain conditions, our scheme emphasizes empirical pragmatism. The shaping terms directly encourage behaviors visible in task-solving trajectories, making them interpretable to human observers. This alignment between reward terms and human task understanding is central to the explainability objective.

The environment wrappers process observations sequentially: action space modification (RemoveDropActionWrapper) $\rightarrow$ reward

shaping (DoorKeyRewardWrapper) → observation transformation (FullyObsWrapper, FlatObsWrapper). This ordering ensures that reward shaping operates on the raw environment state before observation transformation, preserving the causal connection between agent actions and reward signals.

4 Adversarial Callback Architecture

4.1 Overview of the Feedback Mechanism

The adversarial callback implements LLM-driven feedback integrated into the PPO training loop. Rather than applying critiques post-hoc, the callback generates penalties during training that immediately influence policy updates. This design embodies the principle of interactive learning: the agent receives not just rewards but explanations of its shortcomings.

The callback operates on a fixed schedule: every k training steps (default $k = 25$), 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 proportional to the critique score. This design balances computational cost (frequent API queries are expensive) with temporal relevance through heuristic-based caching and rate limiting.

4.2 State Representation and Heuristic Calculation

The system uses a two-tiered approach to state representation. The `get_textual_state` function converts the raw MiniGrid observation into 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 while preserving task-relevant information. By focusing on abstracted progress indicators rather than full trajectory descriptions, the system achieves efficient caching and faster API response times.

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:
    - Agent's Current Objective ('key', 'door', or 'none')

- Agent's Progress since last check ('moved_closer', 'moved_further', or 'no_change')
- Agent's Last Action (e.g., '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 focusing on objective-progress-action tuples rather than full trajectories, the prompt reduces token consumption while maintaining evaluation accuracy.

The complete prompt template implemented in `build_adversary_prompt` is shown below:

This prompt template uses Python f-string formatting to inject the current `objective`, `progress`, and `action` values into the prompt at runtime. The JSON format specification and example ensure consistent response parsing, while the structured rubric provides clear evaluation criteria aligned with task semantics.

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 is encountered again (e.g., the agent repeats the same objective-progress-action pattern), the stored critique and penalty score are reused without additional API calls. This design dramatically reduces token consumption for repetitive behaviors while maintaining consistency in penalty application.

The cache grows during training as new objective-progress-action combinations are encountered. Cache hits are logged for monitoring, 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

Load the Phase 1 checkpoint and attach the `AdversarialCallback` with $k = 25$ (checking every 25 steps) and $\lambda_{\text{penalty}} = 1.0$ (full penalty scaling). Fine-tune the agent for an additional 100,000 timesteps, now receiving both shaped rewards and LLM-generated penalties. A new logger targeting `ppo_adversarial_finetuned_logs/` captures metrics from this phase. The fine-tuned model is saved as `ppo_adversarial_finetuned.zip`.

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 validation 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

Visual inspection allows researchers to observe whether agent behavior correlates with logged critiques and to identify failure modes not captured by aggregate metrics.

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 displays the goal completion rates for both the baseline and fine-tuned agents. The adversarial fine-tuned model achieves a substantially higher success rate of 98.5% compared to the baseline adversarial model trained from scratch, which reaches 59.0%. This dramatic improvement of approximately 39.5 percentage points demonstrates the effectiveness of LLM-driven adversarial feedback in refining agent policies during the fine-tuning phase.

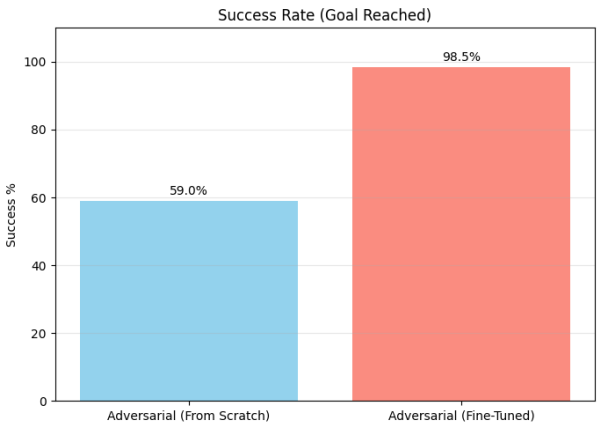

Figure 2: Success Rate (Goal Reached) comparison between adversarial agents trained from scratch versus fine-tuned. The fine-tuned approach achieves 98.5% success rate.

6.2 Reward Progression Over Training Episodes

Figure 3 illustrates the total reward accumulated per episode across training. The blue line represents the moving average of rewards for the

baseline agent trained from scratch, while the orange line represents the fine-tuned agent. The baseline agent exhibits significant variability in early training with generally negative rewards, oscillating between approximately −20 and 0. In contrast, the fine-tuned agent (orange line) rapidly stabilizes at or near 0, indicating consistent achievement of task objectives with minimal penalties.

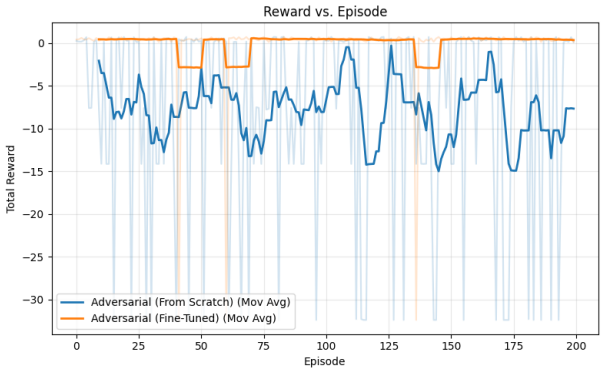

Figure 3: Total Reward vs. Episode showing moving averages for both training approaches. Fine-tuned agent demonstrates superior reward stability.

6.3 Inference Time per Episode

Figure 4 presents the wall-clock inference time measured in seconds per episode. The baseline agent exhibits substantial variability, with inference times ranging from approximately 0.05 to 0.25 seconds per episode. The fine-tuned agent, by contrast, maintains consistently low inference times, typically below 0.02 seconds, with rare spikes. This consistent, efficient performance aligns with the smoother trajectory patterns observed qualitatively and suggests that the LLM feedback successfully eliminated oscillatory behaviors that consume additional timesteps.

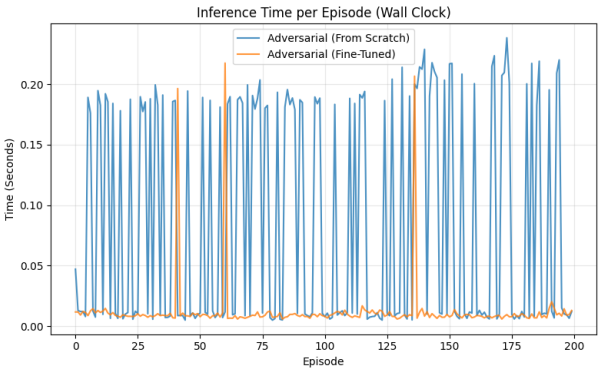

Figure 4: Inference Time per Episode (Wall Clock) demonstrating the computational efficiency of fine-tuned policies.

6.4 Mean Steps to Completion

Figure 5 compares the mean number of steps required to complete the task. The baseline agent requires an average of 109.2 steps (with substantial variability shown by error bars), while the fine-tuned agent completes the task in approximately 14.3 steps on average. This sevenfold reduction in trajectory length represents a major improvement in policy efficiency and directly reflects the impact of adversarial LLM feedback in eliminating wasteful, oscillatory behaviors near the key and door locations.

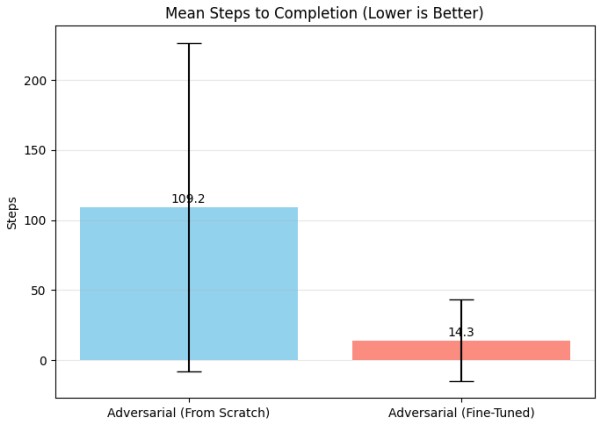

Figure 5: Mean Steps to Completion (Lower is Better) with error bars indicating variability. Fine-tuned agent achieves substantial efficiency gains.

6.5 Qualitative Trajectories

Figures 6 and 7 provide visual renderings of representative episodes from each training approach. These trajectory visualizations serve as qualitative validation of the quantitative metrics. The baseline trajectory often exhibits wandering behavior and suboptimal navigation patterns, while the fine-tuned trajectory demonstrates direct, purposeful navigation to the key followed by efficient movement to the door and goal location.

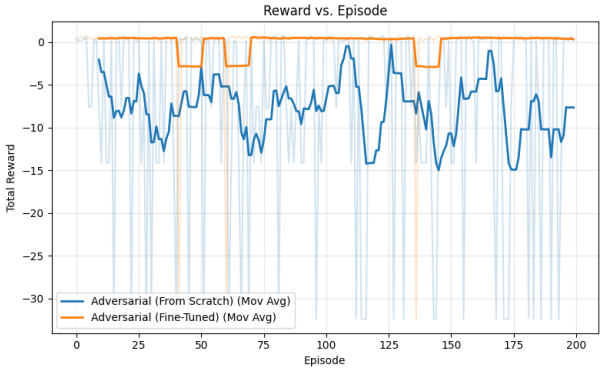

Figure 6: Representative trajectory visualization from baseline agent, illustrating oscillatory behavior and suboptimal navigation patterns.

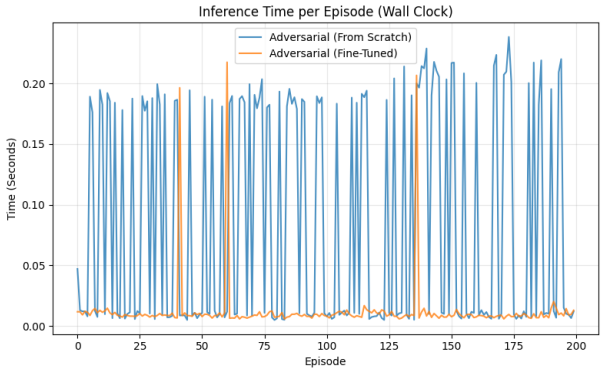

Figure 7: Representative trajectory visualization from fine-tuned agent, demonstrating direct, efficient pathfinding and goal-directed behavior.

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 Explainability Artifacts and Analysis

7.1 Types of Explainability Artifacts

The training process produces multiple categories of explainability artifacts:

7.1.1 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.

7.1.2 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.

7.1.3 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.

7.1.4 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.

7.1.5 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.

7.1.6 Aggregate Statistics

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

7.2 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.

7.3 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.

8 Limitations and Risk Assessment

8.1 Experimental Limitations

The current implementation exhibits several limitations:

8.1.1 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.

8.1.2 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.

8.1.3 Prompt Brittleness

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

8.1.4 Manual Hyperparameter Tuning

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

8.2 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

9 Lessons Learned and Conclusions

9.1 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.

9.2 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.

9.3 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 learn-

ing

- 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 Glossary

PPO Proximal Policy Optimization, a gradient-based RL algorithm

MiniGrid A suite of minimalist grid-world environments for RL research

LLM Large Language Model, used here for generating critiques

Reward Shaping Technique to augment rewards for faster learning

Rollout Buffer Structures used by PPO to store transitions for policy updates

Critique Textual feedback generated by Gemini based on heuristic inputs (objective, progress, action)

Penalty Score Numeric value between 0.0 and 1.0 determining adversarial penalty strength

Cache Key Tuple (objective, progress_str, action_str) used to cache LLM critiques

Heuristic-based Evaluation Evaluation method using abstracted state features (objective, distance) rather than full trajectories

Fully Observable Wrapper Exposes the entire grid to the agent

Flat Observation Wrapper Converts multi-dimensional observations to flat vectors

Deterministic Policy Agent action selection without stochastic sampling during evaluation

B Configuration Snapshot

C File Manifest

Primary project files:

- `training.py` — Main training loop with baseline and fine-tuning phases
- `adversarial_callback.py` — LLM-driven callback with heuristic-based caching
- `eval.py` — Evaluation scripts for model comparison
- `requirements.txt` — Python dependencies
- `.env` — API key management (not version controlled)

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

Log directories:

- `ppo_scratch_logs/` — Phase 1 baseline training logs
- `ppo_adversarial_finetuned_logs/` — Phase 2 fine-tuning logs (if fine-tuning is run)

Model checkpoints:

- `ppo_adversarial_from_scratch.zip` — Baseline trained model (500k timesteps)
- `ppo_adversarial_finetuned.zip` — Fine-tuned model (additional 100k timesteps with adversarial callback)

D 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.