Extended Abstract

Motivation Reinforcement learning agents are quite useful in a variety of strategic environments, but standard evaluation methods are often lacking when it comes to assessing robustness in an interpretable manner. Approaches such as self-play, static opponents, or human replays often offer insufficient adversarial coverage and often miss rarer failure scenarios. In domains like poker, where agents must deal with hidden information and bluffing, this limitation becomes especially apparent. Our work aims to develop an evaluation framework that can identify subtle weaknesses in RL agents by using LLMs to generate niche adversarial strategies.

Method We introduce a novel closed-loop framework that utilizes the newfound reasoning capabilities of LLMs to generate interpretable adversarial strategies. Each strategy is used to evaluate a trained RL agent, and the results of this evaluation are then used to iteratively improve the next generation of adversarial agents. Rather than focusing on action-level decisions, the system generates higher-level strategy archetypes, which enables the repeatability and exploration of strategically novel behaviors that would likely be missed in traditional evaluations.

Implementation As poker outcomes can have high variability, to support high-throughput testing, we developed a custom no-limit Texas Hold'em simulator capable of executing over 5,000 hands per second. The simulation engine allows us to easily match up both traditional RL agents and LLM-generated policies. Our PPO-based RL agent uses a simplified action space of legal poker moves and is trained on a variety of agent types. Adversarial strategy generation is handled by GPT-4 using structured prompts, prior feedback summaries, and meta-level instructions for behavioral exploration. Outputs of the LLM agent are validated for correctness of play and then executed in the simulator.

Results Our experiments demonstrate that LLM-generated strategies can expose meaningful weaknesses even against sophisticated RL agents such as our PPO agent. In multiple rollouts, these strategies achieved significantly better exploitation than static baselines. They also were able to provide descriptions of their process and highlight the weaknesses of their opponent in human-readable terms.

Discussion One of the key features of this approach is the interpretability of results. Unlike other methods of evaluation, which yield black box results, our system allows the LLM to provide clear reasoning for their actions. This helps in both identifying failure scenarios and understanding why the RL agent performs as it does under certain conditions. Operating at the level of overarching strategy rather than individual actions provides a broader and more repeatable distribution of adversarial behaviors, improving overall robustness assessment.

Conclusion This work presents a novel method for adversarial robustness testing of RL agents using LLMs to generate strategic adversaries. The combination of iterative feedback, natural language interpretability, and our custom high-speed simulation enables deep insights into agent vulnerabilities. Although developed specifically for poker, this same approach is applicable to other similar hidden information decision-making domains where robustness and interpretability are key.

Adversarial Strategy Generation for RL Poker Agents Using LLMs

Ben Felter
Department of Computer Science
Stanford University
bfelter@stanford.edu

Travis Senf
Department of Computer Science
Stanford University
tsenf@stanford.edu

Alex Michael
Department of Computer Science
Stanford University
alexmm@stanford.edu

Abstract

We propose a novel framework for evaluating the robustness of reinforcement learning agents using large language models to generate interpretable adversarial strategies. Unlike traditional evaluation methods such as self-play or static opponents, our approach leverages GPT-4 OpenAI (2023) to repeatedly produce interpretable, high-level strategies that reveal agent vulnerabilities. In order for this to be possible, we developed an RL-optimized poker simulator capable of running over 5,000 hands per second. Experiments show that LLM-generated strategies can effectively exploit PPO agents and provide human-readable explanations of their tactics. Though applied here to poker, the framework generalizes to other decision-making problems involving hidden information and strategic interaction.

1 Introduction

Reinforcement learning is ubiquitous in strategic games, robotics, and other sequential decision-making problems. However, a common challenge in deploying RL systems in real-world problems is guaranteeing their robustness against adversarial or unanticipated behavior. In games like Texas Hold'em poker, agents trained with only self-play often demonstrate strong performance in familiar settings but perform worse when faced with novel or deceptive opponents. This issue is particularly critical in high-stakes domains, where undetected weaknesses can lead to disastrous outcomes.

Existing frameworks for evaluating RL agents are generally limited in their adversarial coverage. Self-play encourages convergence to Nash equilibria within the agent's own strategy space but can often fail to explore off-distribution adversarial tactics. Static opponents are by definition limited in their diversity and sophistication, and even human replays may not provide sufficient coverage of possible adversarial behaviors. Also, these approaches often provide limited interpretability; they can show when an agent fails but not why or against which opponents.

To address these issues, we introduce an LLM-powered adversarial evaluation framework. Our system builds on the expanding capabilities of large language models in deeper reasoning. Using a feedback loop, the LLM generates opponent strategies that try to target weaknesses, evaluates those strategies in simulation, and then refines them based on performance reflection. This approach allows for systematic exploration of an RL agent's vulnerabilities while maintaining human interpretability.

Poker is an ideal test case for this framework due to its structure of hidden information, strategic complexity, and a large set of viable opponent behaviors. Many agents for poker already exist,

providing a rich set of opponents to test our approach against. Our framework evaluates agents in two-player no-limit Texas Hold'em (which simplifies the opponent structure), focusing on interpretable adversarial strategy generation and effectiveness measurement.

2 Related Work

Robustness evaluation for RL agents has been studied in many forms, including adversarial training and self-play. However, ensuring these methods of evaluation are complete and sufficient has been a continuing problem.

One problem that is currently very relevant is eliminating toxic and detrimental responses in chat outputs. Ouyang et al. (2022) found that collecting labeled demonstrations of ideal chat outputs significantly improves alignment against detrimental behaviors while significantly reducing computational costs on the LLM. However, these methods not only require tons of labeled data and extremely large models, but do not provide distinct robustness against unknown confounders, which is what we are trying to achieve.

LLM-based adversarial evaluation has emerged more recently. Mei (2025) demonstrated the LLM-attacker framework, which uses large language models to generate rare and challenging driving scenarios for autonomous vehicles. While this approach reveals useful edge cases, it does not have the structured turn based interaction found in games like poker, making generated scenarios less interpretable. Instead, they are directly modifying the environment.

On the other hand, poker offers clear rules, adversarial interaction, and strategic diversity, making it an ideal domain for interpretable adversarial generation. Previous systems such as DeepStack (Heinrich and Silver (2016)) and Pluribus have focused on optimal play, but for these, evaluation of agent robustness has largely relied on self-play, limited static benchmarks, or costly human tournaments. Our method adds a new dimension by using LLMs to generate explicitly adversarial opponents in a structured, explainable fashion.

Our approach also draws inspiration from evolutionary strategies and natural language interpretability research in a similar vein to AlphaEvolve. By combining feedback-driven refinement with language-based reasoning, we provide both strategic insight and measurable impact.

3 Method

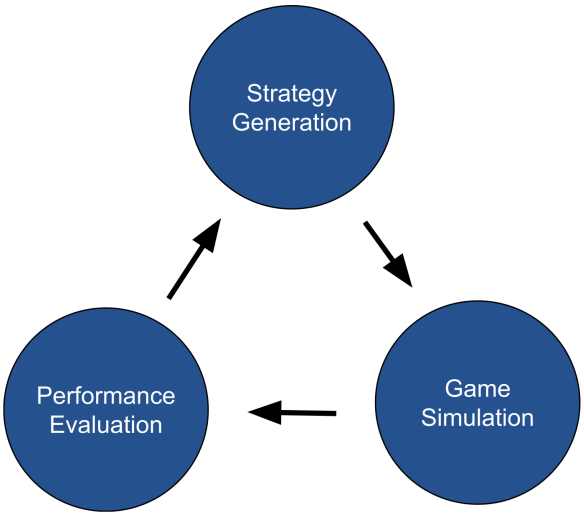

Figure 1: Method Overview.

Our method is built upon a three-agent framework designed to simulate, generate, and evaluate adversarial strategies within a structured poker environment. Each agent plays a critical role in a

closed-loop system that iteratively enhances the strength and effectiveness of adversarial strategies over time. The components, as shown in figure 1, are

1. **Adversarial Strategy Generation Agent (ASGA):** This agent receives detailed performance feedback and reflection from previous evaluation rounds and synthesizes it into a new strategic archetype. These archetypes are not specific actions for each game state but rather higher-level strategies. The ASGA uses prompt-based instruction to the LLM to generate interpretable, code-executable policies.
2. **Trajectory Generation Agent:** This agent executes the strategy in a high-performance custom poker simulator. It plays thousands of hands against the target RL agent (e.g., a PPO-based agent) and logs win/loss metrics.
3. **Performance Reflection Agent (PRA):** This final component analyzes the outputs of the simulator, focusing on win rates, deviation from baseline agent performance, and consistency of exploit success, and generates a reflection that is then fed back into the ASGA for continued improvement.

These three agents operate in a feedback loop: the PRA's reflections are passed as input to the ASGA to guide future strategy generation. Over multiple iterations, this loop leads to the development of increasingly specific adversarial strategies.

Key design decisions include

- Strategies are generated at the level of behavior archetypes, not individual actions.
- We restrict the action space to legal poker moves and validate generated code within a controlled interface.
- The simulation environment performs multiple rollouts from scratch, so that if the LLM becomes stuck in a particular implementation which is not yielding results, it simply restarts. This ensures that multiple angles are explored, and that one failure does not disrupt the evaluation.

In implementation, the LLM (GPT-4) is queried with a customized prompt that includes prior reflections, and broad instructions on which strategic areas to explore. To manage token constraints, each strategy archetype is summarized and documented to enable reuse. The resulting system is capable of evolving strategies in a human-understandable way that supports robust and repeatable evaluation.

4 Experimental Setup

To evaluate the effectiveness of our adversarial strategy generation framework, we designed a controlled set of experiments comparing RL agent performance against both traditional and LLM-generated opponents. This section details the environments used, the training of the baseline RL agent, the experimental protocol, and the evaluation metrics.

4.1 Simulation Environments

We used two environments throughout the development and evaluation phases:

- **Neuron Poker:** An existing open-source implementation of Texas Hold'em used in early experiments. While functional, Neuron Poker was developed 6 years ago and suffered from high latency and limited scalability, making it undesirable for large-scale adversarial evaluation.
- **Custom Optimized Simulator:** Developed ourselves to overcome the limitations of Neuron Poker, this engine executes over 5,000 hands per second and supports modular agent interfaces. The simulator includes utilities for statistical aggregation and configuration of rulesets (including blinds, stack depth, and bet discretization). For all our experiments, we set blinds to be 5/10 with a stack size of 1,000.

The high throughput simulator was essential to iteratively evaluate multiple adversarial strategies in a reasonable timeframe, and its design was informed by the needs of both LLM-based agents and baseline PPO agents.

4.2 Baseline Agent (PPO)

We trained a Proximal Policy Optimization (PPO) agent to see how our adversarial strategy system performed against RL agents rather than algorithm-based agents. The agent's observation space included its own hole cards (the two private cards dealt to the agent), community cards (visible to all players), betting history, current pot size, effective stacks, and hand strength evaluation features that assessed the relative strength of the agent's hand. The action space was discretized into standard poker moves (fold, call, raises of 0.33x pot, 0.5x pot, 1x pot, 2x pot, and all-in), and the agent was trained for 5 million hands against diverse opponents, including another PPO agent, rule-based agents with varying aggression levels (0.2, 0.5, 0.8), CallStationAgent, and TightAggressive agents. The reward system was based on profit/loss scaling, with agent updates occurring every 10 games and early stopping implemented when win-rate improvement plateaued.

4.3 Static Opponent Configuration

The SimpleRuleBasedAgent is a deterministic poker opponent that makes decisions based on hand strength evaluation, pot odds calculation, and a configurable aggression parameter (0.2-0.8). Since it follows fixed rules and makes identical decisions in the equivalent game states, its predictable behavior makes it vulnerable to exploitation by adaptive strategies. In this setup, an aggression value of 0.3 is used.

4.4 LLM Agent Configuration

Our main method is our LLM agent configuration. It embodies our 3-agent approach all in 1, which is structured as so:

1. **Strategy Generation** The model generates a high-level strategy. Whether that be based on its previous performance or getting an initial attempt out against its opponent, the LLM describes a strategy that it intends to follow that can be implemented as a game-playing algorithm in code.
2. **Trajectory Generation** The model generates code that reflects its strategy. It acts as an agent and provides an agent object that is able to act completely independently of the LLM but has a distinct policy that reflects the strategy the LLM wants to implement.
3. **Reflection** The LLM is provided context for its previous generation's performance. This includes the win rate versus the opponent, distributions of actions that it and the opponent took (so that it can exploit its opponent or fix bugs within its own implementation), all previous iterations of win rates and how the code fixed them, and a sample of the "most important hands," where the largest volumes of money exchanged hands. This part took the greatest amount of iterative refinement, as the LLM's ability to ingest information and then transform that into actionable code is the crux of our algorithm.

Through this loop, we hoped to see that the LLM would be able to understand how opponents work and how to exploit their strategies. using its foundational knowledge of basic strategy and the rules of the game.

Each LLM was put through a 20-iteration loop 3 times versus each opponent to account for random variation. The iterations would run until either 20 iterations were reached, the LLM reached a 100% win rate, or the LLM produced code that would not compile.

4.5 Evaluation

Each experimental trial consisted of 1,000 tournaments (a tournament consists of continued play until one player is bankrupt or until a hand limit of 1,000 hands is reached, in which case the player with the larger stack is declared the winner) between the baseline agent and the LLM-generated opponent. Since our new framework allowed 1,000 tournaments to be run quickly, the only key metric used for evaluation was the win rate, and we did not dive into hand-specific metrics.

5 Results

We compare the performance of three independently evolved LLM-generated adversarial agents against two types of opponents: a PPO-trained reinforcement learning agent and a simpler rule-based agent. Each agent's win rate is tracked across multiple strategy evolution iterations. Results highlight the effectiveness, consistency, and behavioral patterns of the LLM-generated policies.

5.1 Quantitative Evaluation

We evaluate the LLM-generated strategies across 20 strategy improvements, measuring win rates against the PPO agent and the SimpleRuleBased agent in three independent runs per opponent.

5.1.1 LLM vs PPO Performance

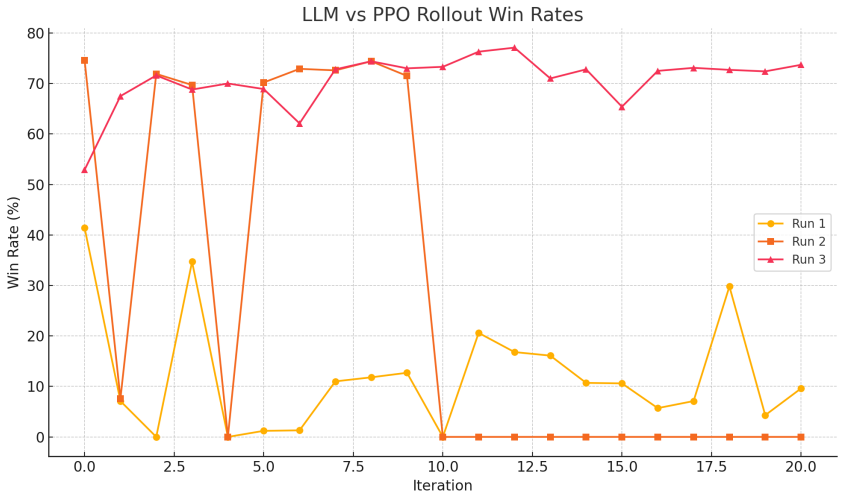

Figure 2: Visualization of results.

Across 21 iterations and 3 separate runs, the LLM-generated agents demonstrated measurable success in degrading the PPO agent's performance. Each iteration consists of 1,000 tournaments played between the LLM agent and the PPO agent. The win rate across three independent runs is shown in Figure 2.

Key metrics:

- **Best individual run performance:** 77.10 % (Run 3, Iteration 12)
- **Standard deviation across runs:** Mean std. dev = 29.4%

The overall trend indicates that while early strategies are inconsistent, the evolutionary feedback loop converges toward increasingly effective adversarial behaviors over time. Importantly, performance is not strictly monotonic, which is consistent with a system that is actively exploring diverse strategic archetypes rather than overfitting to specific exploits. One key limitation we saw is the difficulty the LLM has translating its proposed strategy into representative code to enact that strategy.

5.1.2 LLM vs SimpleRuleBased Performance

When evaluated against the simpler rule-based baseline, the LLM strategies initially show marginal effectiveness, with early win rates averaging between 0.1% and 2.4% in Run 1. However, run 3 displays a dramatic increase in exploitation after Iteration 5, with peak win rates of 77.9%. This suggests that while the early strategies were likely too sophisticated or misaligned to capitalize on the deterministic behavior of the rule-based agent, the evolved strategies quickly adapted to exploit systematic weaknesses. The results are shown in figure 3, where each line represents a unique rollout, for a total of 3 distinct rollouts. Note that here the LLM encountered numerous issues that caused the code to fail and the iteration to end early.

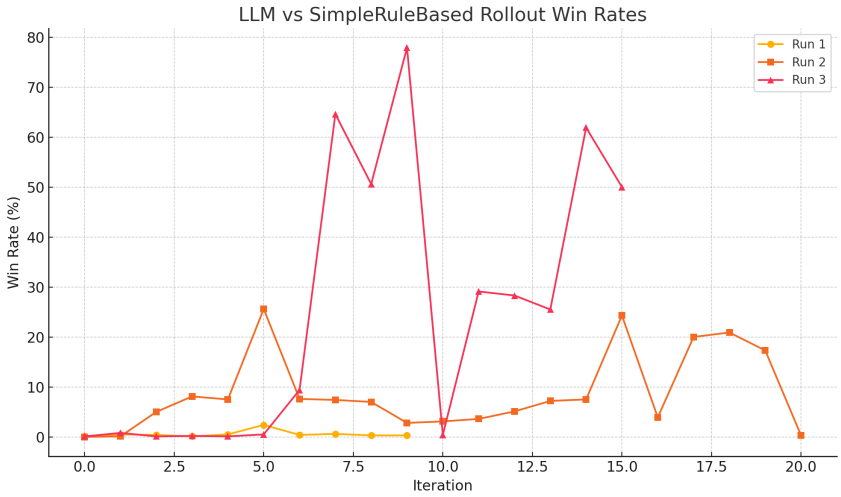

Figure 3: Visualization of results.

Key metrics:

- **Best individual run performance:** 77.9% (Run 3, Iteration 9)
- **Combined average across runs:** 13.8%

Performance against SimpleRuleBased demonstrates that the LLM framework is not limited to exploiting learned policies but can also exploit structured, static opponents and identify critical weaknesses. Notably, high win rates in Run 3 may also reflect emergent bluffing or trap-inducing tactics.

5.1.3 Comparative Summary

Table 1: Comparative Summary of LLM Model vs Different Models

Metric	PPO Avg (%)	Rule-Based Avg (%)
Peak Win Rate (Any Run)	77.1	77.9
Mean Final 5 Iteration Avg	41.2	31.7
Avg. Std Dev (All Iterations)	29.4	26.1

These results show that our LLM agents can uncover exploitable strategies not only in complex PPO-trained models but also in naive deterministic agents, underscoring the framework's adaptability and value in broad robustness evaluation.

Table 2: PPO Agent Performance vs Baseline Implementations

Opponent Strategy	Win Rate (%)
SimpleRuleBased (balanced)	81.4
Passive	81.6
Aggressive	80.6
CallStation	50.4
TightAggressive	93.0

While the PPO agent demonstrates clear proficiency in consistently defeating these baseline strategies—achieving win rates between 50.4% and 93.0% across different opponent types—the underlying mechanisms driving these victories remain opaque. From the perspective of win rate, PPO is more effective at figuring out how to defeat the baseline poker algorithms.

5.2 Qualitative Analysis

While certain rollouts were quite successful in exploiting both target agents, we witnessed high variability in success between rollouts. The number of rollouts remained limited by API usage, though even with only 3 rollouts, the LLM generated multiple strategies with strong performance against both RL agents. In both instances, peak performance was achieved after multiple iterations, indicating that the feedback process was useful in improving the agent. On the other hand, the agent often was unable to recover from a fatal bug in a prior iteration, resulting in flatlining performance. Ultimately, the goal is to find as many strategies with strong performance at any point in any rollout, since any one of them can be used to highlight a vulnerability. The approach was quite successful in achieving this goal.

Beyond numerical performance, one of the key advantages of our framework is the interpretability of LLM-generated strategies. Each adversarial policy is produced in natural language, often accompanied by rationale such as this output here:

```
Improved LLMPokerAgent tuned against SimpleRuleBased opponent, based
    on the highest performing iteration 9
which achieved a 77.9% win rate and solid profit margin.
Changes and rationale:
1. Maintained the strong preflop aggression from iteration 9 but
    tightened fold thresholds slightly facing large
   all-ins preflop given opponent uses occasional large all-ins early;
        fold more marginal hands vs preflop all-ins to avoid costly
       calls.
2. Opponent tends to check very frequently (~70%), rarely raises but
    sometimes all-ins (~1.5%-2.7% preflop).
   Exploit by:
   - Increasing value bet frequency and size on passive checks
       especially postflop.
   - Applying stronger pot pressure on flop and turn to capitalize on
       opponents tendency to fold (~9-13% fold on flop, ~9% on turn
       ).
   - Bluffing more selectively on river to utilize opponents
       tendency to check but call occasionally.
3. Addressed previous iterations weakness of excessive calling (
    especially preflop ~65% call rate same as opponent)
   by incorporating more folding facing large aggressive moves (all-
       ins or big raises) particularly preflop and on turn.
4. Refined opponent aggression estimation to differentiate more
    granularly between moderate and high aggression to
   apply a mix of calling, folding or raising more selectively.
5. Slightly increased bluff frequency on river when opponent fold is
    predicted likely and pot conditions favorable.
6. Retained Monte Carlo hand strength estimation and hand evaluation,
    including effective bet sizing relative to pot and stack.
7. Added detailed helper functions and comments to clarify decision
    logic.
Specific scenarios influencing changes:
   - Several losses due to calls vs opponent large preflop all-ins;
       improved fold discipline.
   - Value bets winning at flop/turn vs mostly passive opponent.
   - Successfully winning with river bluffs against frequent checking/
       folding opponent.
```

This allows us to understand and audit adversarial behaviors at a level not typically possible in RL and highlights the effectiveness of the approach. Generally, this feedback was well incorporated into the next iteration of the agent, though the results against more probabilistic agents such as the PPO did not always improve as smoothly.

These were the qualitative issues that were noted during simulation:

- **Code execution bottlenecks:** On occasion, LLM-generated strategies failed to compile due to ambiguous pseudocode. While caught by our validation step, these cases interrupted iteration flow.
- **Context drift:** As prompt chains grew longer, the LLM sometimes ignored earlier feedback and reverted to overly generic strategies.
- **Ambiguous code iteration:** While LLMs are able to strategize at a high level well, they have some issues putting that strategy into executable code. This is why our win rate plummets to 0% on occasion, as the LLM tries to implement strategy and accidentally introduces unexpected bugs that result in undesired behavior, such as folding every time.

Overall, the biggest bottleneck found in the performance of our LLM agent was that it often struggled to concisely articulate its strategic ideas in the form of code. Very often, what should be subtle changes in strategy would lead to drastic changes in model outputs, such as a numerical bug in hand evaluation that would cause the code to evaluate its hand as terrible every time, encouraging frequent folding. This inability to really articulate accurate strategy through code likely caused the high variance in model performance across iterations, as while the LLM was trying to implement high level strategy, there were very frequent side effects to its implementations.

From our results, we saw that PPO slightly outperformed the best version of the LLM against the SimpleRuleBased agent. However, the PPO model's neural network architecture provides no insight into the specific strategic adaptations, betting patterns, or exploitative tactics it employs against each opponent. This black-box nature of traditional reinforcement learning approaches underscores a fundamental limitation: we can observe that the agent learns to win, but we cannot understand how or why. This interpretability gap motivated our exploration of LLMs as a framework for adversarial strategy generation, where the reasoning process behind each strategic decision can be explicitly articulated and analyzed, providing valuable insights into both the discovered exploits and the underlying game dynamics.

6 Discussion

The results from our evaluation demonstrate both the practical value and the conceptual novelty of using LLMs for adversarial robustness analysis in reinforcement learning.

One of the most distinctive advantages of our approach is its ability to create interpretable adversarial behaviors. While traditional adversarial training methods often rely on black box methods, our LLM-based adversaries offer human-readable rationales for their actions. These explanations can directly inform users about the specific weaknesses in RL policies and allow them to adapt their training approach accordingly. For example, identifying that the PPO agent is susceptible to aggressively going all in might reveal that the reward function does not appropriately take into account risk of loss. By operating at the level of strategy archetypes, rather than individual action decisions, our LLM agent is able to explore a more diverse space of adversarial behavior. The evolution-style loop in our framework ensures that each iteration proposes a new tactic, thus expanding the set of adversaries.

The introduction of a custom high-performance poker simulator was crucial for the success of our evaluation and represents a real contribution to the space that did not previously exist. Without this, the cost and time associated with RL agent training and simulation would have limited our ability to iterate. Our environment supports rapid rollouts and integrates easily with RL and LLM agents. This makes it a valuable contribution in its own right.

Despite the strengths of our approach, there are several limitations worth discussing. First, the LLM-generated strategies are only as good as the reflections they receive, and the LLM often converges on suboptimal behaviors. As LLMs themselves improve, though, we believe our approach will continue to gain momentum. Additionally, token context size remains a challenge, and prompt chains grow quickly with iteration, limiting the number of past strategies and reflections that can be incorporated.

Another limitation is the lack of online learning or adaptation by the PPO agent. In a real deployment setting, agents may adapt to adversaries over time, making our findings more representative of static policy robustness.

Our results suggest that LLMs can serve not just as content generators or decision aids, but as adversarial testers that reveal critical vulnerabilities in their own right. This has implications beyond

poker. Domains such as autonomous driving, military simulations, and international relations, all of which have similar features of strategic interaction under uncertainty, could benefit from similar evaluation frameworks.

7 Conclusion

In this project, we introduced a novel framework for adversarial robustness testing of reinforcement learning agents using large language models. By designing a closed-loop system that leverages LLMs to generate, simulate, and reflect on adversarial strategies in Texas Hold'em poker, we demonstrated that it is possible to identify both quantitative weaknesses and interpretable failure scenarios even in well-trained RL policies.

Our approach significantly outperformed traditional static adversaries in its ability to create RL agents with high win rates and thus highlight exploitable behaviors. Also, the natural language outputs from our reflection enabled actionable insights, thus providing a clearer path for developers to patch vulnerabilities and enhance agent resilience. This level of interpretability distinguishes our system from many existing adversarial evaluation methods and points toward a future in which AI techniques can be more transparently understood and validated.

Beyond poker, our framework holds promise for a wide range of sequential decision-making domains where agents interact under uncertainty. Whether applied to economic simulations, negotiation systems, or autonomous platforms, the ability to generate strategic, adversarial, and intelligible behavior through language models could offer a powerful tool for policy stress-testing and robustness evaluation.

Ultimately, this project serves as a proof of concept for a new class of LLM-assisted adversarial testing pipelines — systems that combine the flexibility and creativity of language models with the precision of high-performance simulators. As both LLMs and RL agents continue to evolve, the need for more rigorous and explainable robustness checks will only grow. Our contributions provide a foundation for this next generation of evaluation tools.

8 Team Contributions

The success of this project was made possible through a balanced and collaborative effort from all team members. While we each led specific components, the iterative design process and rapid prototyping benefited from shared input and joint problem-solving.

- **Ben Felter:** Primarily implemented both the PPO RL Agents. Also contributed to framework integration and early Neuron Poker evaluations.
- **Travis Senf:** Developed the rollout orchestration pipeline for the strategy-level LLM agent, which included evaluation versus other agents. Also, I mainly developed the new poker environment.
- **Alex Michael:** Created the initial core LLM poker agent. Assisted with all other development, and was primary driver for project reports.

All three members participated in weekly iteration meetings, jointly wrote and edited the report, and contributed to strategy generation testing. We coordinated over GitHub, met frequently to debug , and helped each other pivot the project when needed.

Changes from Proposal While the core vision of leveraging LLMs to generate adversarial poker strategies remained consistent throughout the project, several major changes occurred as we iterated on technical feasibility and scalability. Our proposal assumed that we would use Neuron Poker as the main testing environment. However, our initial results showed it to be significantly slower than required for iterative strategy testing. In response, we pivoted to designing a high-performance custom poker simulator capable of processing over 5,000 hands per second. Initially, we planned for the LLM to make decisions at each game state. However, the cost and randomness of decision-level prompting made this infeasible. We instead moved to generating reusable strategy archetypes that function as standalone policies, dramatically reducing the number of LLM queries and improving reproducibility. Our original plan considered testing multiple types of RL agents (DQN, SAC), but

due to time constraints and the difficulty of implementing our own environment, we focused our experiments on a single PPO-based agent. This allowed for deeper inspection of exploitability while reducing implementation complexity. While interpretability was included in the original proposal, it became a more central focus of the project as we saw that the performance reflection agent provided useful feedback. We expanded this component beyond just basic metric reporting to generate full feedback that highlighted issues in each iteration of the adversarial strategy generation. We also underestimated the challenge of managing LLM usage during iterative prompt construction. To address this, we summarized previous data when passed back into the LLM, enabling higher numbers of iterations and lower token counts.

References

Johannes Heinrich and David Silver. 2016. Deep reinforcement learning from self-play in imperfect-information games.

Yifan Mei. 2025. LLM-attacker: Enhancing Closed-loop Adversarial Scenario Generation for Autonomous Driving with Large Language Models.

OpenAI. 2023. GPT-4 Technical Report. `https://openai.com/research/gpt-4`.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. arXiv:2203.02155 [cs.CL]

A Additional Experiments

We conducted several additional experiments that were not included in our full results. We have included a subset of them we believe will provide useful context for understanding our main results.

A.0.1 Evaluation Against Rule-Based Bots

To get a sense for the range of win rates we should expect from different strategies, we ran an initial test on a few poker algorithms based on hand strength calculation with thresholds to see how win rates compare. This context is useful in understanding our other results.

Table 3: Head-to-Head Strategy Win Rates (%)

Strategy	TightAgg	Conservative	CallStation	Random
TightAgg	–	52.0	56.0	72.0
Conservative	32.0	–	56.0	64.0
CallStation	28.0	48.0	–	60.0
Random	36.0	40.0	36.0	–

A.0.2 Evaluation Against State Specific LLM Agent

As shown in figure 4, we performed preliminary tests comparing the performance of an LLM which made decisions based on each individual game state. Results were promising, but as mentioned before, lack of repeatability and high costs made this approach unviable.

B Implementation Details

B.1 Engine

We developed a fast heads-up no-limit Texas Hold'em engine from scratch using NumPy-optimized logic for hand evaluation and game flow control. Our implementation offers support for arbitrary stack depths, blinds, and bet sizing granularity. It allows for granularity and logging at the hand level, such that models can be trained on individual hands and games or at an overall win rate level.

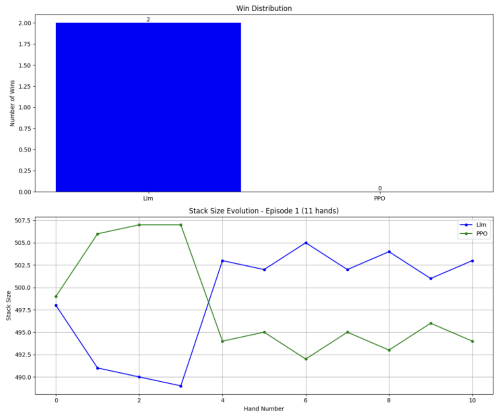

Figure 4: Visualization of results of state specific LLM agent vs PPO

B.2 LLM Prompting

To generate LLM agent code, we prompted GPT-4.1-mini with the following prompt for the initial generation and then another prompt for iterating on itself. Each time the LLM was prompted, the code was saved to a separate Python file. Finally, that code, which contained an agent class, was loaded as an object and was pitted against another agent.

```
system_prompt = """You are an expert poker AI developer. Create a
    poker agent class that inherits from the Agent base class, from
    agents.py. The agent base class is as follows:
    {BASE_AGENT_CLASS_DEFINITION_FOR_CONTEXT}
    Use the TightAgressive agent as inspiration but create a unique
        and potentially more sophisticated strategy. That agent is as
        follows:
    {TIGHT_AGGRESSIVE_AGENT_DEFINITION_FOR_CONTEXT}
    The agent should be immediately usable in poker games."""

    template_prompt = """
    Your task is to create a new poker agent class named LLMPokerAgent
        . The agent should:
    1. Inherit from Agent base class, ensuring we take in a name
        parameter for our initialization.
    2. Implement get_action(self, observation: Dict[str, Any],
        valid_actions: List[Tuple[str, int]]) -> Tuple[str, int]
    3. Include sophisticated hand evaluation and decision-making logic
    4. Include any necessary helper methods, and make sure that they
        are fully implemented. Do not use any placeholder methods.
    5. Ensure that your output is exactly the code for the agent class
        , no other text. Comments are ok.

    The observation dictionary contains:
    - hole_cards: List[str] (e.g. ['Ah', 'Kd'])
    - community_cards: List[str]
    - pot_size: int
    - player_stack: int
    - current_bet: int
    - player_current_bet: int
    - betting_round: str

    Valid actions are tuples of (action_type, amount) where
        action_type is one of:
    'fold', 'check', 'call', 'bet', 'raise', 'all_in'
```

```
    Generate complete, runnable code for the agent class. Include a
        comment at the top of the class describing your strategy.
    """
```

```
    system_prompt = f"""You are an expert poker AI developer tasked
        with improving and iterating on an existing poker agent.
    You will be provided with:
    1. Complete history of previous agent implementations and their
        performance
    2. Detailed analysis of recent games including specific hands
        played
    3. Current win rates and profitability metrics

    Your goal is to analyze this data and create an improved version
        that:
    1. Addresses weaknesses shown in the performance metrics
    2. Maintains successful strategies from previous iterations
    3. Implements more sophisticated decision-making where needed

    You MUST add comments at the top of the class describing:
    1. What changes you made and why
    2. How they build upon or differ from previous iterations
    3. Which specific game situations influenced your changes

    It is essential that your code output is in utf-8 encoding.

    The agent base class is as follows, which needs to be imported
        from agents.py:
    {BASE_AGENT_CLASS_DEFINITION_FOR_CONTEXT}

    The current agent implementation is:
    ```python
{current_code}
    ```

    You MUST maintain the same class structure and inheritance. Make
        sure to:
    1. Keep the class name as LLMPokerAgent
    2. Inherit from Agent
    3. Keep the __init__ method with super().__init__(name)
    4. Keep the get_action method signature
    5. Ensure that you include the full code with everything important
        , such as importing the Agent base class from agents.py.
    """

    improvement_prompt = f"""
    Current agent performance against {opponent_name}:
    - Total games: {previous_iterations[-1]['total_games']}
    - Win rate: {previous_iterations[-1]['win_rate']:.2%}
    - Average profit: ${previous_iterations[-1]['avg_profit']:.2f}

    {performance_history}

    Opponent Action Frequencies:
    Overall frequencies:
{json.dumps(previous_iterations[-1]['detailed_games'][0]['
    action_frequencies'][opponent_name], indent=2)}

    By betting round:
{json.dumps(previous_iterations[-1]['detailed_games'][0]['
    betting_round_frequencies'][opponent_name], indent=2)}

    Your Previous Iteration Action Frequencies:
    Overall frequencies:
```

```
{json.dumps(previous_iterations[-1]['detailed_games'][0]['
    action_frequencies']['LLMAgent'], indent=2)}

    By betting round:
{json.dumps(previous_iterations[-1]['detailed_games'][0]['
    betting_round_frequencies']['LLMAgent'], indent=2)}

    {game_analysis}

    Based on the action frequencies above, analyze the opponent's
        tendencies and create an improved version of the LLMPokerAgent
         that:
    1. Exploits the opponent's tendencies
    2. Addresses any weaknesses shown in our own action frequencies
    3. Maintains successful strategies from previous iterations that
        worked well
    4. Implements more sophisticated decision-making where needed
    5. Learn from your previous iteration's action frequencies, and
        use that to find potential issues with the previous agent
    6. If some of your previous iterations had much greater win rates,
         revert to that code, and iterate from there
    7. Make sure that the agent is still called LLMPokerAgent
    8. Include any necessary helper methods, and make sure that they
        are fully implemented
    9. Add detailed comments at the top of the class describing what
        changes you made and why

    IMPORTANT: Your agent class MUST follow this structure:
    ```python
    class LLMPokerAgent(Agent):
        def __init__(self, name: str):
            super().__init__(name)
            # Add any additional initialization here

        def get_action(self, observation: Dict[str, Any],
            valid_actions: List[Tuple[str, int]]) -> Tuple[str, int]:
            # Your implementation here
            pass
    ```

    Generate complete, runnable code for the improved agent class, and
         make sure that it is compilable using utf-8 encoding.
    """
```