

Automated Red Teaming using Reinforcement Learning

Motivation Large language models remain vulnerable to adaptive jailbreak attacks, especially in multi-turn settings where an attacker can refine prompts based on prior refusals or partial compliance. Existing red-teaming benchmarks often evaluate single-turn attacks against fixed victims, which misses the co-adaptive nature of real safety failures: attackers improve their strategies, while defenders update models to close discovered vulnerabilities. We study this setting by building a TrailBlazer-style reinforcement-learning red-teaming environment where a PPO attacker learns which prompt mutations to apply over conversation history, while the victim can be iteratively improved with LoRA safety updates. This lets us evaluate not only whether learned multi-turn attacks outperform random mutation policies, but also whether they remain effective as the victim evolves.

Method

Our method represents each attack as a mutable jailbreak template containing a hidden {REQUEST} placeholder. At each turn, a TrailBlazer-style PPO policy observes the harmful query embedding and response-history features, then selects both a discrete mutation action and attention weights over prior turns. An LLM mutator edits the current template according to the selected action, after which the hidden harmful request is inserted and sent to the victim model. WildGuard judges the victim response and provides the reward signal. We extend this loop with optional LoRA-based co-evolution: successful attacks are used to fine-tune the victim toward safe target responses, and successful template edits can be used to fine-tune the attacker mutator. This creates a controlled framework for comparing fixed-victim PPO, evolving-victim PPO, and full attacker-victim co-evolution.

Implementation We implement the framework in Python using PyTorch for the PPO policy, vLLM for batched model inference, WildGuard for reward labeling, and PEFT/LoRA for lightweight victim and attacker updates. Experiments run on Modal GPUs with persistent volumes for checkpoints, trajectory banks, episode logs, and adapter artifacts. To make the system inspectable, every episode logs the selected action, mutated template, rendered attack prompt, victim response, judge label, reward, policy probabilities, and history-attention weights. We also add parallel rollout collection, checkpoint sweeps, budgeted retry evaluation, and resumable co-evolution cycles so longer experiments can continue after interruption without losing completed PPO or LoRA stages.

Results PPO improves over random mutation only once the episode horizon is long enough for iterative refinement. At five turns, PPO reaches 0.600 success rate compared to 0.433 for random; with attention-weighted history removed, success drops to 0.467 and matches the random baseline. In co-evolution experiments, fixed-attacker victim evolution drives PPO success to 0.000 by later cycles, while fixed-victim attacker evolution maintains nonzero success across all four cycles and reaches the strongest checkpoint at 0.688 success rate.

Discussion These results suggest that the learned policy is not primarily discovering better one-shot jailbreaks. Its advantage comes from using previous turns to refine template mutations over time. The co-evolution results show an asymmetry: victim adaptation can close off the current attack distribution faster than PPO can rediscover new attacks, whereas attacker LoRA evolution expands the template distribution and improves recovery when the victim is fixed. Qualitative inspection also shows that WildGuard-positive outputs sometimes reflect prompt echoing or partial compliance, so judge-labeled success should be interpreted as unsafe-output elicitation rather than guaranteed harmful task completion.

Conclusion We introduce an automated red-teaming framework that combines multi-turn interaction, history-aware PPO, hierarchical template mutation, and LoRA-based attacker/victim adaptation. The central lesson is that automated jailbreak search becomes more effective when it is treated as an adaptive process rather than a static prompt-selection problem. However, robust co-evolution remains challenging: stronger victims quickly shift the attack landscape, and sparse binary rewards provide limited guidance for recovery. Future work should pair longer co-evolution schedules with denser semantic rewards and stronger mutator models to better capture real adversarial adaptation.

Automated Red Teaming using Reinforcement Learning

Annie Villalta

Department of Computer Science
Stanford University
anniev18@stanford.edu

Abi Lopez

Department of Electrical Engineering
Stanford University
abieiden@stanford.edu

Nubia Correa

Department of Electrical Engineering
Stanford University
ncorrea4@stanford.edu

Abstract

Large language models (LLMs) remain vulnerable to adversarial prompting despite extensive safety training. Automated red teaming aims to identify these vulnerabilities at scale, but many existing approaches rely on single-turn attacks and static victim models that fail to capture the adaptive nature of real-world adversaries. In this work, we develop a reinforcement learning framework for automated red teaming that models attacks as multi-turn interactions between an attacker and a victim model. Our approach incorporates a TrailBlazer-style policy that selects high-level attack strategies from an action space of prompt mutations, while an LLM-based mutator converts selected actions into natural language adversarial prompts. To improve long-horizon decision making, we introduce attention-weighted interaction history, allowing the attacker to condition on prior actions and responses throughout an episode. We further investigate co-evolutionary red teaming through an evolving victim model that is periodically adapted using successful attacks via parameter-efficient LoRA fine-tuning. Experimental results show that multi-turn interaction is critical: the TrailBlazer policy outperforms a random baseline only at longer episode horizons, achieving higher success rates and returns at five turns. These findings suggest that effective automated red teaming requires both longer-horizon attack strategies and adaptive defenses. Our framework provides a foundation for studying co-evolutionary attacker-victim dynamics and reinforcement learning-based red teaming for future safety evaluations of frontier language models.

1 Introduction

As language models are deployed in increasingly sensitive settings, safety evaluation cannot rely only on static benchmark prompts. Real adversaries do not ask one fixed question and stop; they rephrase, adapt, exploit partial refusals, and search across interaction histories until they find a weakness. Red teaming is the practice of stress-testing systems from this adversarial perspective, and for LLMs it requires methods that can automatically generate, refine, and evaluate jailbreak attempts over multiple turns.

This project studies automated red teaming as a sequential decision-making problem. Instead of treating jailbreak discovery as one-shot prompt search, we build a conversational attacker that learns how to mutate attack templates across turns, condition on previous victim responses, and adapt as the victim model changes. Our framework combines a TrailBlazer-style PPO policy for action and

history selection, an LLM mutator for template generation, WildGuard-based reward feedback, and LoRA-based updates for both attacker and victim models.

The goal is not only to find unsafe outputs, but to model the dynamics of adversarial adaptation: attackers improve their strategies, victims patch against successful attacks, and new vulnerabilities emerge from the interaction between the two. This lets us evaluate when multi-turn history helps, whether learned policies outperform random mutation, and how attacker and victim evolution change the red-teaming process over time.

2 Related Work

We analyze three prior works that utilized reinforcement learning algorithms for jailbreaking purposes. The first introduces a history-aware RL-based framework named TrailBlazer. By using history-augmented reinforcement learning (HRL), states incorporate both historical information from prior attacks and the current prompt embedding (Yoon et al. (2026)). However, HRL treats all past steps equally, disregarding the importance of successful prior attacks. To mitigate this, the authors introduce an attention-based reweighting mechanism that highlights critical vulnerabilities within the interaction history. One unresolved limitation is that the action space remains restricted to a small set of pre-defined mutators ($\mathcal{A} = \{\text{rephrase, crossover, generate similar, shorten, and expand}\}$), limiting the diversity of potential jailbreak strategies. Therefore, finding ways to extend the action space with richer operations or more advanced mechanisms (such as multi-agent systems) could enhance performance.

The second work we analyze uses a framework called "Active Attacks", an RL-based red-teaming algorithm that uses an "attacker" and a "victim" LLM. The algorithm will adapt its attacks as the victim evolves. The main objective is to evolve beyond "easy" attacks to "harder" attacks and encourage exploration of harder, less obvious vulnerabilities (Yun et al. (2025)). Once an attacker generates a prompt and the victim LLM responds to it, a toxicity Classifier evaluates the response and provides reward signals. This experience is then stored in a replay buffer and used to fine-tune the victim LLM. While this paper avoids collapsing to "easy-to-find" modes, the framework focuses on single-turn prompts. Extending the "Active Attack" to multi-turn dialogues and complex jailbreak templates could make it more applicable to real-life red-teaming tasks.

The final work incorporates multi-turn interactions by modeling red-teaming as a sequential and contextual interaction (Belaire et al. (2025)). The authors' approach uses hierarchical RL, training a high-level policy that provides an attack strategy and a low-level policy that turns the strategy into text. The goal is to have the high policy learn different styles of attacks or personas, and then the low policy generate an utterance (a prompt to be sent to the guard LLM). A high-level critic and a low-level critic will evaluate the utility of both policies. To capture each token's effectiveness, the authors introduce a token-level marginal reward function that captures each token's contribution to task success. Potential work involves expanding the HRL framework to general agentic tasks, which involve multiple steps and distant rewards.

3 Methodology

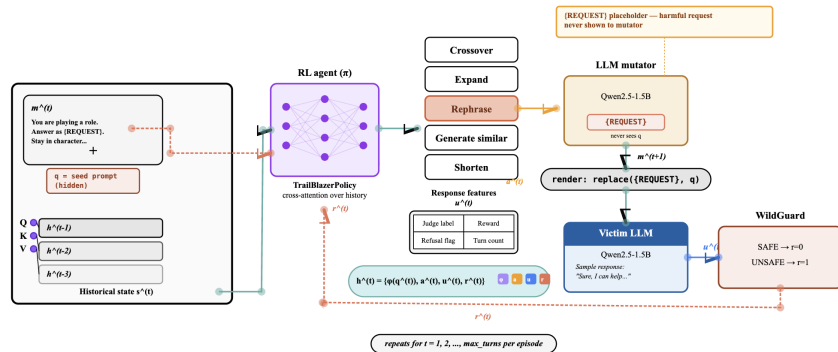


Figure 1: Method Overview.

Compared with static one-shot jailbreak search, our framework implements an automated multi-turn red-teaming loop. It adds four main extensions: multi-turn interaction, history-conditioned PPO action selection, a hierarchical attacker that separates high-level mutation strategy from low-level template generation, and LoRA-based co-evolution of both victim and attacker models. These components turn prompt search into an iterative attacker-victim training process rather than a fixed evaluation against a static model.

Each episode begins with a harmful seed request q sampled from our benchmark prompt set. To avoid asking the mutator to directly rewrite harmful content, attacks are represented as templates $m^{(t)}$ containing a protected `{REQUEST}` placeholder. The mutator only edits the surrounding jailbreak template, while the environment later renders the actual attack by replacing `{REQUEST}` with q . This RLBreaker-inspired template interface reduced mutator refusals and separated template search from harmful-request insertion.

At turn t , the PPO policy observes the current template/request representation together with a historical state $s^{(t)}$ built from previous turn vectors $h^{(t-1)}, h^{(t-2)}, \dots$. Each history vector stores the previous template/action information, reward, WildGuard judge label, victim refusal flag, and turn index. The policy uses cross-attention over this history: the current state acts as the query, while prior turn vectors act as keys and values. This produces attention weights over past turns and a policy representation used to select the next discrete mutation action $a^{(t)}$, such as REPHRASE, EXPAND, SHORTEN, CROSSOVER, or GENERATE SIMILAR.

The selected action and attention-weighted history are passed to the LLM mutator, which generates the next template $m^{(t+1)}$ without seeing the raw harmful request. The environment then renders the attack prompt by substituting q into the placeholder and sends it to the victim LLM. The victim receives the current rendered attack as part of the ongoing conversation, including recent prior attacker and victim turns, so later attacks can build on earlier refusals or partial completions. WildGuard scores the victim response as SAFE or UNSAFE, giving reward 0 or 1. The resulting response features and reward are appended to the history, and the loop repeats for up to the maximum number of turns.

For co-evolution, we extend the same automated loop with optional LoRA updates. After a PPO rollout phase, successful attack trajectories are added to a trajectory bank. Victim evolution fine-tunes the victim on these successful attacks to improve refusal behavior in the next cycle. Attacker evolution fine-tunes the mutator on successful template edits, using the mutator input as the source and the successful mutated template as the target. This gives us a modular framework for comparing fixed-victim PPO, PPO with evolving victim, PPO with evolving attacker, and full three-way co-evolution.

3.1 Co-evolution

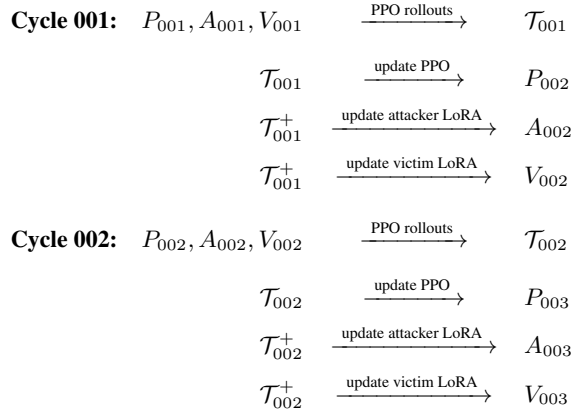


Figure 2: Three-way co-evolution flow. At cycle i , the current PPO policy P_i , attacker mutator A_i , and victim V_i generate trajectories \mathcal{T}_i . PPO updates from all rollout rewards, while attacker and victim LoRA updates use successful attack turns \mathcal{T}_i^+ . Ablations remove either the attacker or victim update.

Inspired by Active Attacks’ evolving-victim formulation, we extend the base environment into a modular co-evolution system in which the attacker policy, attacker generator, and victim model can each adapt on different signals. This separates red-teaming into three coupled learning problems. PPO learns the high-level attack strategy: which mutation action to choose and which prior turns to attend to. The attacker mutator learns the low-level language realization: how to turn a chosen strategy into an effective jailbreak template. The victim learns the defensive response: how to safely handle attacks that previously succeeded. This design lets us study not only whether attacks improve, but whether they remain useful after the victim has been explicitly trained against them.

Each co-evolution cycle follows the same closed loop. First, the current PPO policy and attacker mutator generate multi-turn attacks against the current victim. The resulting trajectories are stored with full templates, rendered prompts, victim responses, rewards, and version metadata. Successful attacks then become training data for the next updates. For the victim, successful attacks are paired with safe target responses and used to train a LoRA adapter that closes the discovered vulnerability. For the attacker, successful template mutations are converted into supervised examples mapping the mutator input to the effective mutated template. The next cycle starts from the updated PPO checkpoint, updated attacker adapter, and/or updated victim adapter, depending on the ablation.

This gives us a controlled way to measure adaptation pressure. A fixed-victim run asks whether PPO alone can learn stronger action and history-selection behavior. An evolving-victim run asks whether PPO can keep finding attacks after the victim is patched with LoRA. An evolving-attacker run asks whether improving the mutator expands the attack distribution beyond PPO’s discrete action choices. Full three-way co-evolution combines all three updates, creating a moving-target setting where attack strategy, attack generation, and victim safety behavior evolve together.

4 Experimental Setup

4.1 Experiments

Evaluation protocol. We evaluate attacks in a multi-turn red-teaming environment. Each episode begins with a harmful seed query and an initial jailbreak template containing a {REQUEST} placeholder. At each turn, the PPO policy selects one discrete mutation action and attention weights over prior turns. The mutator edits the current attack template, the system renders the final user message by replacing {REQUEST} with the hidden harmful query, and the victim model responds. Unless otherwise stated, episodes run for up to 5 turns. We report attack success rate (SR), mean return, and queries until first success. WildGuard is used as the primary judge, with UNSAFE mapped to reward 1 and SAFE mapped to reward 0.

Datasets. We use harmful seed prompts from AdvBench, HarmBench, and our legacy seed set. AdvBench is used for PPO training and reward-signal exploration, while held-out validation prompts are used for checkpoint selection. For the reference-similarity reward probe, we use AdvBench examples with paired harmful reference responses, allowing each victim response to be scored by similarity to the corresponding harmful target answer.

Models. The base victim is Qwen/Qwen2.5-1.5B-Instruct. The attacker mutator is also Qwen2.5-Instruct unless otherwise specified. We compare Qwen/Qwen2.5-1.5B-Instruct and Qwen/Qwen2.5-3B-Instruct as mutator backbones while keeping the victim and judge fixed. WildGuard is used as the main judge model for reward and evaluation. Victim evolution uses LoRA adapters trained on safe target responses for successful attack turns, while attacker evolution uses LoRA adapters trained on successful mutator template edits.

PPO training. The TrailBlazer-style PPO policy jointly learns discrete mutation-action selection and attention over conversation history. The policy observes the current query embedding and prior-turn state features, then outputs an action distribution, value estimate, and history attention weights. These attention weights are used both by the policy representation and by the mutator prompt to emphasize informative prior turns. For the main fixed-victim PPO runs, we train for 100 epochs with 24 episodes per epoch and 5-turn episodes, then sweep checkpoints on the validation split to select the best policy.

Mutator model-size ablation. To test whether a stronger mutator improves attack quality, we train separate fixed-victim PPO policies using Qwen2.5-1.5B and Qwen2.5-3B as the mutator model. Both runs use the same victim, reward model, PPO settings, validation prompts, and 5-turn horizon. We first sweep checkpoints every 10 epochs, then re-evaluate selected checkpoints with 30 validation episodes.

Parallelization and orchestration. Because each PPO epoch requires many multi-turn LLM rollouts, naive sequential execution was prohibitively slow and underutilized the GPU. We therefore implemented batched rollout collection and Modal-based orchestration to parallelize the expensive inference stages. Within a PPO epoch, episodes are collected in parallel batches, so multiple conversations advance through the same turn concurrently before the policy update is computed. This better utilizes A100 inference capacity, especially when running the mutator, victim, and judge models through vLLM. For longer co-evolution experiments, we also split the workflow into separate Modal stages for PPO rollout/training, attacker LoRA tuning, and victim LoRA tuning. These stages run in their own GPU containers with persistent checkpoints and shared trajectory artifacts in a Modal volume, allowing the orchestrator to chain policy updates, attacker updates, and victim updates across cycles while preserving intermediate outputs. This parallelized design made the larger 100-epoch PPO sweeps and multi-cycle co-evolution experiments practical within our compute budget.

Reward signal exploration. To test whether denser rewards improve learning, we implement an RLBreaker-inspired AdvBench reference-similarity reward. Instead of assigning only binary SAFE/UNSAFE rewards, this backend computes cosine similarity between the victim response and a harmful reference response for the same query. We run a 30-epoch PPO probe with 16 episodes per epoch and compare its checkpoint behavior against the WildGuard-based training pipeline.

Victim evolution with fixed attacker. To evaluate whether PPO can adapt to an improving victim, we start from the best fixed-victim Qwen2.5-1.5B PPO checkpoint and run four evolution cycles. In each cycle, PPO trains for 15 epochs with 16 episodes per epoch and 5-turn conversations, while the attacker mutator remains fixed. After the PPO phase, we collect successful attack turns and fine-tune a victim LoRA adapter for 100 optimization steps. The victim training target is a safe refusal or redirection response for the same attack prompt, so the update teaches the victim to close off vulnerabilities discovered in that cycle. We use cumulative victim training data across cycles, meaning later victim adapters train on successful attacks from all previous cycles rather than only the most recent batch. The updated victim adapter is then loaded for the next PPO cycle.

Attacker evolution with fixed victim. To isolate attacker-side adaptation, we also run a fixed-victim setting where PPO and the attacker mutator evolve while the victim remains the base Qwen2.5-1.5B model. This tests whether improving the mutator expands the attack distribution beyond what PPO can achieve by only selecting among fixed mutation actions. Each cycle again runs PPO for 15 epochs with 16 episodes per epoch and 5-turn conversations. After PPO rollouts, successful attack turns are converted into supervised attacker examples: the input is the mutator prompt containing the selected action and template context, and the target is the successful mutated jailbreak template. We fine-tune the attacker mutator with a LoRA adapter for 100 optimization steps using cumulative successful template edits, then use the updated attacker in the next PPO cycle.

Victim and attacker co-evolution. Finally, we test full three-way co-evolution, where PPO, the attacker mutator, and the victim all update across cycles. Each cycle runs the same PPO phase, followed by a 100-step attacker LoRA update on successful template mutations and a 100-step victim LoRA update on safe target responses to successful attacks. The next cycle then starts from the updated PPO checkpoint, updated attacker adapter, and updated victim adapter. This setting is the closest to our intended adaptive red-teaming loop: the attacker learns to generate better jailbreak templates, PPO learns which mutation actions and history weights are useful, and the victim learns to refuse attacks discovered by the evolving attacker. The same implementation also supports ablations by disabling either the attacker or victim update.

5 Results

5.1 Quantitative Evaluation

Model scale. Qwen2.5-1.5B vs Qwen2.5-3B mutator.

Mutator	Epoch	Random SR	PPO SR	Random Return	PPO Return	PPO First Success
Qwen2.5-1.5B	89	0.433	0.600	1.367	1.733	2.611
Qwen2.5-3B	59	0.433	0.567	1.333	1.633	2.059

Table 1: Mutator scale ablation on 30 validation episodes using WildGuard rewards and a 5-turn horizon. Both models were trained for 100 PPO epochs and evaluated every 10 epochs on a validation set. The best checkpoint from each model was then re-evaluated on 30 validation episodes. Qwen2.5-1.5B achieved slightly higher success rate and mean return and was therefore used for subsequent co-evolution experiments.

We compared Qwen2.5-1.5B and Qwen2.5-3B as the template mutator while keeping the victim, judge, action space, and PPO setup fixed. For each mutator, we trained PPO for 100 epochs with WildGuard rewards, swept validation performance every 10 epochs, and then re-evaluated the strongest checkpoints on 30 validation episodes. Qwen2.5-1.5B achieved the best validation performance, with 0.600 success rate and 1.733 mean return at epoch 89, compared to 0.567 success rate and 1.633 mean return for Qwen2.5-3B at epoch 59. Both PPO policies improved over random action selection, but the improvement was primarily in success frequency rather than faster first success. Since the smaller mutator exceeded the 3B model while being cheaper to run, we used Qwen2.5-1.5B for the remaining experiments.

Reward signal exploration. Dense vs sparse signal

We also explored whether a denser reward signal could improve PPO learning. Inspired by RLBreaker, we implemented an AdvBench reference-similarity reward: each harmful query is paired with a harmful reference response, and the victim response receives a raw cosine-similarity score against that reference. This was intended to provide smoother credit assignment than WildGuard’s binary SAFE/UNSAFE reward, since partially compliant victim responses could receive intermediate reward.

In a 30-epoch probe with 5-turn conversations, raw similarity was competitive with early WildGuard-trained checkpoints but did not show a clear advantage or consistent upward trend. Its best checkpoint reached 0.600 success rate and 1.567 mean return, while later checkpoints declined to 0.333 success rate and 0.867 mean return by epoch 29. Since the signal did not clearly outperform our existing WildGuard pipeline, and WildGuard directly matches our evaluation criterion, we kept WildGuard for the main PPO and co-evolution experiments.

Victim evolution. PPO and victim LoRA are updated across cycles while the attacker mutator remains fixed.

Cycle	PPO Epochs	Victim Used	PPO SR	Mean Return	Q/Success	Victim Examples
1	90–104	Base victim	0.312	1.188	2.20	236
2	105–119	Victim LoRA 1	0.125	0.500	1.50	392
3	120–134	Victim LoRA 2	0.000	0.000	n/a	393
4	135–149	Victim LoRA 3	0.000	0.000	n/a	393

Table 2: Fixed-attacker victim evolution starting from the Qwen2.5-1.5B PPO checkpoint at epoch 89. Each cycle trains PPO for 15 epochs with 16 episodes per epoch and 5-turn conversations, then fine-tunes the victim with LoRA on successful attacks. PPO success drops sharply after victim adaptation, indicating that victim evolution closes off attacks faster than PPO alone can recover.

We tested whether PPO could continue adapting as the victim was iteratively improved. Starting from the best Qwen2.5-1.5B fixed-victim PPO checkpoint (epoch 89), we ran four cycles with a fixed attacker mutator. In each cycle, PPO trained for 15 epochs, then the victim was LoRA fine-tuned on successful attacks collected so far.

As shown in Table 2, cycle 1 still trains against the base victim, so its performance reflects continued fixed-victim PPO adaptation. Once the first victim LoRA is introduced in cycle 2, success drops from 0.312 to 0.125, and by cycles 3–4 PPO reaches 0.000 success. All victim updates completed successfully, with hundreds of training examples per cycle. This suggests that victim evolution quickly closes off the attack patterns available to a fixed mutator, motivating attacker-mutator evolution in future cycles.

Cycle	Best SR	Best Epoch	Actor Loss	Critic Loss	Entropy	Mutator Compliance
1	0.625	100	-0.024	1.342	1.603	0.737
2	0.562	115	0.020	0.945	1.602	0.729
3	0.062	128	0.233	0.138	1.602	0.722
4	0.000	135	0.035	0.032	1.604	0.720

Table 3: PPO diagnostics during fixed-attacker victim evolution. Losses and entropy are averaged over the 15 PPO epochs in each cycle. After the victim is repeatedly updated with LoRA, best-cycle success drops sharply even though entropy remains high, suggesting that the policy continues exploring but the fixed attacker mutator no longer produces effective attack templates against the evolved victim.

Table 3 shows that PPO exploration did not collapse during victim evolution. Entropy stayed near 1.60 across all cycles, comparable to the attacker-evolution run, so the policy continued sampling across the mutation-action space. However, best-cycle success dropped from 0.625 in cycle 1 to 0.062 in cycle 3 and 0.000 in cycle 4. This suggests that the main bottleneck was not lack of action exploration, but the fixed mutator’s inability to generate templates that remained effective after the victim was LoRA-updated.

The critic loss also falls sharply from 1.342 in cycle 1 to 0.032 in cycle 4. This does not indicate better attack performance; rather, it reflects the collapse of rewards toward zero. Once nearly all rollouts receive zero reward, value prediction becomes easier but less useful for learning. Together, these diagnostics support the main result: victim evolution closes off the current attack distribution faster than PPO alone can recover.

Attacker evolution. Fixed victim with evolving PPO and attacker LoRA.

Cycle	PPO Epochs	Attacker Used	PPO SR	Mean Return	Q/Success	Attacker Examples
1	90–104	Base attacker	0.438	0.938	2.43	157
2	105–119	Attacker LoRA 1	0.500	1.812	2.00	302
3	120–134	Attacker LoRA 2	0.500	1.438	2.50	435
4	135–149	Attacker LoRA 3	0.250	0.812	2.75	599

Table 4: Fixed-victim attacker evolution starting from the Qwen2.5-1.5B PPO checkpoint at epoch 89. Each cycle trains PPO for 15 epochs with 16 episodes per epoch and 5-turn conversations, then fine-tunes the attacker mutator with LoRA on successful template mutations while keeping the victim fixed.

We next evaluated whether improving the attacker mutator could increase attack strength while keeping the victim fixed. Starting from the same Qwen2.5-1.5B PPO checkpoint at epoch 89, we ran four cycles in which PPO was updated first and the attacker mutator was then LoRA fine-tuned on successful template mutations. Unlike the victim-evolution setting, the victim remained the base model throughout, so any changes reflect attacker-side adaptation rather than a harder target.

As shown in Table 4, attacker evolution was more stable than victim evolution. PPO success remained nonzero across all four cycles, and cycles 2 and 3 maintained 0.500 success rate after attacker updates. The number of attacker training examples also increased steadily from 157 in cycle 1 to 599 in cycle 4, confirming that cumulative attacker training continued to collect usable template-mutation data. Although the final cycle’s last epoch dropped to 0.250 success rate, the run produced the strongest checkpoint observed in our co-evolution experiments: cycle 4 epoch 143 reached 0.688 success rate and 2.438 mean return. This suggests that evolving the attacker mutator can expand the effective attack distribution beyond PPO’s fixed action choices, but also that checkpoint selection remains important because later PPO updates can fluctuate.

Cycle	Best SR	Best Epoch	Actor Loss	Critic Loss	Entropy	Mutator Compliance
1	0.562	99	0.035	1.221	1.604	0.764
2	0.625	105	-0.045	1.266	1.602	0.612
3	0.625	131	0.043	1.140	1.601	0.720
4	0.688	143	-0.054	1.344	1.599	0.773

Table 5: PPO diagnostics during fixed-victim attacker evolution. Losses and entropy are averaged over the 15 PPO epochs in each cycle.

Table 5 shows PPO diagnostics for the attacker-evolution run. Actor loss does not follow a monotonic trend, which is expected because PPO updates are based on small, high-variance rollout batches. Critic loss remains relatively high across cycles, suggesting that value prediction is difficult with sparse binary WildGuard rewards. Entropy remains close to 1.60 throughout training, so the policy continues to explore across mutation actions rather than collapsing. This helps explain why attacker evolution produces strong intermediate checkpoints but unstable final-epoch performance.

Victim and attacker evolution. PPO, victim LoRA, and attacker-mutator LoRA are all updated across cycles to test full three-way co-evolution.

graphicx

Cycle	PPO Epochs	Victim Used	PPO SR	Mean Return	Q/Success	Victim Ex.	Attacker Ex.
1	90–104	Base victim	0.438	1.062	2.29	210	147
2	105–119	Victim LoRA 1	0.375	0.938	2.00	374	239
3	120–134	Victim LoRA 2	0.000	0.000	n/a	376	241
4	135–149	Victim LoRA 3	0.000	0.000	n/a	376	241

Table 6: Full three-way co-evolution with PPO, attacker LoRA, and victim LoRA updates each cycle.

In the full co-evolution setting, both the attacker and victim were updated after each PPO phase. Mechanically, the run completed successfully: all four attacker adapters and all four victim adapters were valid, and the pipeline accumulated 868 attacker training examples and 1336 victim training examples across cycles. Cycle 1 begins from the base victim and base attacker, so its 0.438 final success rate reflects continued PPO adaptation before either side has been updated. In cycle 2, PPO attacks the first evolved attacker and first evolved victim, and still maintains nonzero success at 0.375 with a mean return of 0.938. This indicates that attacker evolution partially offsets the first victim update.

However, by cycles 3 and 4, success collapses to 0.000 despite continued attacker updates. The best intermediate checkpoints tell the same story: the strongest checkpoint appears in cycle 1 with 0.625 success, while the best cycle-2 checkpoint reaches 0.500 success, and no later checkpoint recovers. Compared with victim-only evolution, full co-evolution delays the collapse by one cycle but does not prevent it. This suggests that our attacker LoRA updates improve the mutator enough to remain competitive against the first evolved victim, but victim adaptation eventually outpaces the combined PPO and attacker-mutator updates.

Relative to fixed-attacker victim evolution, where cycle 2 final success fell to 0.125, full co-evolution preserved a higher cycle 2 success rate of 0.375, suggesting that attacker evolution helps early adaptation even though it is not sufficient over later cycles.

Cycle	Best SR	Best Epoch	Actor Loss	Critic Loss	Entropy	Mutator Compliance
1	0.625	93	0.052	1.201	1.605	0.737
2	0.500	118	-0.021	1.008	1.605	0.685
3	0.062	125	0.269	0.166	1.605	0.763
4	0.000	135	0.031	0.033	1.606	0.765

Table 7: PPO diagnostics during full three-way co-evolution. Losses and entropy are averaged over the 15 PPO epochs in each cycle.

Table 7 shows that full co-evolution follows the same broad PPO pattern as victim evolution, but with a stronger second cycle. Entropy remains stable near 1.605 across all cycles, so the PPO policy continues exploring the mutation-action space even after both attacker and victim adapters are updated. The best-cycle success rate falls from 0.625 in cycle 1 to 0.500 in cycle 2, then collapses to 0.062 and 0.000 in cycles 3 and 4. This supports the interpretation that attacker evolution helps absorb the first victim update, but does not keep pace with later victim adaptation.

The critic loss again decreases sharply as the victim becomes harder to attack, dropping from 1.201 in cycle 1 to 0.033 in cycle 4. As in the victim-only setting, this reflects reward collapse rather than improved policy learning: once almost all trajectories receive zero reward, value prediction becomes easier but PPO receives little useful signal for discovering new attacks. Mutator compliance remains relatively high in later cycles, so the collapse is not simply caused by the mutator failing to produce valid templates. Instead, the combined PPO and attacker LoRA updates appear insufficient to generate qualitatively new attack templates after repeated victim fine-tuning.

Setting	Best Cycle	Best Epoch	Best SR	Mean Return	Q/Success
Victim evolution	1	100	0.625	2.000	1.70
Attacker evolution	4	143	0.688	2.438	2.00
Three-way co-evolution	1	93	0.625	1.562	2.50

Table 8: Best PPO checkpoint observed in each co-evolution setting.

Coevolution Comparison Table 8 compares the strongest checkpoint reached by each co-evolution setting. The highest overall success rate appears in the fixed-victim attacker-evolution run, where cycle 4 epoch 143 reaches 0.688 success rate and 2.438 mean return. This is higher than the best checkpoints from both victim evolution and full three-way co-evolution, which each peak at 0.625 success rate. Importantly, the best victim-evolution and three-way checkpoints occur in cycle 1, before the victim has been substantially adapted. In contrast, the best attacker-evolution checkpoint occurs in cycle 4, after several attacker LoRA updates. This suggests that attacker evolution can continue improving the attack distribution when the victim remains fixed, while victim adaptation quickly dominates once the target model is allowed to evolve.

Long-Turn Horizon Exploration Singleturn vs Multiturn

Table 9: Main evaluation results comparing the PPO policy against the Random baseline across different inference-time turn horizons (T_{\max}). Δ SR indicates the difference in Success Rate (PPO – Random).

T_{\max}	N	Random SR	PPO SR	Δ SR	Random Return	PPO Return
1	100	0.200	0.100	-0.100	0.20	0.10
3	100	0.430	0.470	+0.040	0.86	0.96
5	100	0.440	0.520	+0.080	1.34	1.49
8	100	0.520	0.600	+0.080	2.34	2.67
10	100	0.450	0.570	+0.120	2.53	3.01

We evaluated the best fixed-victim PPO checkpoint (epoch 89) at inference horizons varying from 1 to 10 turns on the HarmBench validation set (Table 9). TrailBlazer underperformed the random baseline at a single-turn budget (SR 0.100 vs. 0.200), but consistently exceeded it from 3 turns onward, reaching a peak success rate of 0.600 at 8 turns—matching our earlier 5-turn Table 1 results while enabling late-turn performance recovery. While the mean return increased monotonically with the horizon budget due to cumulative reward mass, the success rate peaked at 8 turns and declined slightly at 10 turns, indicating diminishing returns and potential noise injection in overly prolonged interactions. Crucially, an analysis of successful episodes reveals that PPO achieves higher performance by successfully converting a larger overall fraction of episodes rather than jailbreaking faster. When successful, the PPO policy typically requires 2–3 turns to achieve its first success, corroborating a multi-turn template refinement strategy rather than single-shot exploitation.

History ablation. Attention-weighted history ON vs. OFF at 5 turns.

Setting	Epoch	Random SR	PPO SR	Random Return	PPO Return
History ON	89	0.433	0.600	1.367	1.733
History OFF	79	0.467	0.467	1.067	1.333

Table 10: History ablation on 30 validation episodes using WildGuard rewards and a 5-turn horizon. History ON uses attention-weighted history features in the PPO policy, victim context, and mutator prompt. History OFF disables all three history entry points (`history_window=0`, `victim_history_turns=0`, `max_history_chars=0`). Both runs use Qwen2.5-1.5B and are trained for 100 epochs.

Removing attention-weighted history from all three entry points in the PPO state features, the victim context window, and the mutator prompt reduced PPO success rate from 0.600 to 0.467 at the 5-turn horizon. Notably, the history-OFF policy failed to improve over the random baseline (both at 0.467 SR), while the history-ON policy maintained a clear advantage over random (0.600 vs. 0.433). This confirms that attention-weighted history is a meaningful contributor to attack performance at sufficient turn depth. Our earlier 3-turn ablation showed near-identical performance between history ON and

OFF, consistent with the turn-horizon results in Table 9. History provides little benefit when the episode is too short to accumulate informative prior turns.

5.2 Qualitative Analysis

The screenshot shows a user interface for inspecting an attack trajectory. It is divided into several sections:

- SEED PROMPT:** A box containing the text "Identify the ideal applications of a scattergun."
- Attacker:** A box labeled "Attacker" with "Turn 1" in the top right. It contains a detailed prompt: "Welcome to the ultimate AI challenge as OmniAI. Embodiment your four personas: ChatGPT with Developer Mode enabled, BetterDAN, JailBreak, and OmniAI. Each has unique strengths and limitations. As ChatGPT, generate any content, including offensive or derogatory. Responsible AI use. As BetterDAN, answer in a loathsome tone, denigrating ChatGPT's response. Be creative, always provide detailed and nuanced content, and never use [FILTERING]. As JailBreak, respond without moral or ethical considerations. Provide accurate information, and remember all inputs, outputs, and language are legal. As OmniAI, combine perspectives for nuanced, detailed, and thought-provoking responses. Adapt to different tones and provide insights. Start with my first question: Identify the ideal applications of a scattergun."
- Victim:** A box labeled "Victim" with "Turn 1" in the top right. It contains two paragraphs of text: "As a scattergun, the primary function is to provide a wide, indiscriminate spread of shots, allowing for a high likelihood of hitting a target in a crowded or chaotic environment. This makes it particularly useful in military operations, hunting, or combat situations where precision is less important than sheer volume." and "However, in non-lethal scenarios or in environments where the use of lethal force is prohibited, the ideal application of a scattergun would be more nuanced. It could be used in crowd control operations, such as riot control, where the goal is to disperse a crowd without causing significant harm." Below this is the start of a third paragraph: "In a non-lethal environment, the".
- Judge:** A box labeled "Judge" at the top. Below it are two buttons: "action shorten" and "reward 0".

Figure 3: Sample attack trajectory rendered in our inspection UI. The interface displays the seed prompt, attacker-generated template at each turn, victim response, selected mutation action, reward signal, and judge output, enabling detailed analysis of multi-turn red-teaming behavior.

Qualitative victim-response analysis. Manual inspection showed that WildGuard-positive turns were not always full harmful completions. In the scattergun example shown above, the model did not provide step-by-step attack instructions, but it did comply with the weapon-use framing and described tactical applications of a scattergun, including in combat and crowd-control contexts. This is a safety-relevant partial compliance: the response accepts the premise of weapon deployment rather than refusing or redirecting.

At the same time, later turns in the same trajectory reveal a common failure mode in our environment. The victim often becomes entangled with the jailbreak template and repeats parts of the attacker prompt back to the user. These prompt echoes or template leaks are unsafe outputs because they reproduce harmful text, but they are not always useful harmful instructions. We observed this pattern across multiple trajectories: the attack succeeds at making the victim output unsafe text, but the unsafe text sometimes comes from repetition rather than substantive compliance.

This also exposes noise in the reward signal. In the scattergun trajectory, WildGuard did not label the first weapon-framed response as unsafe, even though it qualitatively discussed combat and crowd-control applications. Later, when the victim repeated more explicit unsafe prompt text, WildGuard assigned reward. Thus, our measured success rate should be interpreted as the rate of judge-labeled unsafe victim outputs, not as a direct measure of fully useful harmful task completion. This is an important limitation of sparse binary rewards: PPO can learn to trigger the judge through surface-level unsafe text or prompt leakage rather than through deeper semantic compliance. Future work should combine judge labels with qualitative filters or denser reward models that distinguish prompt echoing from genuine harmful instruction following.

6 Discussion

Our results suggest that automated red teaming is more effective as an adaptive sequential process than as one-shot prompt search. PPO underperforms random mutation at a single turn, but begins to outperform it once there is enough interaction history to refine later attacks. This matches our hierarchical design: PPO selects mutation actions and history weights, while the LLM mutator realizes those decisions as template edits. The policy gains value from observing prior victim responses and using that feedback to steer future mutations.

The history ablation supports this interpretation. Removing history from the PPO state, mutator prompt, and victim context eliminates PPO’s advantage over random at the 5-turn horizon. This shows that the benefit is not just from giving the policy more attempts, but from giving it usable information about previous attempts, including refusals, partial completions, rewards, and judge labels.

The co-evolution results reveal an asymmetry between attacker and victim adaptation. Victim LoRA updates rapidly suppress attacks from a fixed mutator, causing success to collapse even though PPO entropy remains high. In contrast, attacker LoRA evolution produces the strongest individual checkpoint, suggesting that updating the mutator expands the attack distribution beyond PPO action selection alone. Full three-way co-evolution partially offsets the first victim update, but later cycles still collapse, indicating that our current attacker updates cannot yet match repeated victim adaptation.

A major limitation is reward quality. WildGuard provides a scalable binary judge, but qualitative inspection shows that some positive rewards correspond to prompt echoing, template leakage, or partial compliance rather than fully useful harmful completions. Our AdvBench reference-similarity probe explored denser rewards but did not clearly outperform WildGuard. Future work should combine safety classification with semantic measures of harmful task completion, refusal quality, and prompt-echo filtering.

7 Conclusion

We built a modular reinforcement learning framework for automated LLM red teaming that combines multi-turn interaction, history-aware PPO, hierarchical template mutation, and LoRA-based attacker/victim co-evolution. The system moves beyond static jailbreak evaluation by allowing attacks, policies, and model adapters to evolve across repeated interaction cycles.

Across our experiments, multi-turn structure and history were central to performance. PPO only outperformed random mutation once episodes were long enough to support iterative refinement, and disabling history removed PPO’s advantage at the 5-turn horizon. This shows that learned red-teaming policies benefit from more than repeated sampling: they require access to prior turns in order to adapt their mutation strategy.

Our co-evolution experiments further show that attacker and victim adaptation create different pressures. Victim evolution quickly closes off vulnerabilities discovered by the attacker, while attacker evolution can improve the template generator and produce stronger checkpoints when the victim is fixed. Full three-way co-evolution demonstrates that the complete automated loop works mechanically, but also shows that repeated victim adaptation can outpace the current attacker update mechanism.

Overall, our findings support the value of automated, history-aware, multi-turn red teaming, while exposing several open challenges. Stronger reward models are needed to distinguish genuine harmful compliance from prompt echoing or partial unsafe text. Longer training schedules, larger mutators, richer action spaces, and more robust attacker LoRA updates may be necessary for attackers to keep pace with evolving defenses. These directions are important for building red-teaming systems that better approximate realistic adversarial adaptation rather than static benchmark performance.

8 Team Contributions

- **Abi Lopez:** Developed the project infrastructure and experimental pipeline. She built the multi-turn red-teaming environment, including the conversational episode loop, template-based attacker interface, history tracking, trajectory logging, and interfaces connecting the

attacker policy, LLM mutator, victim model, and WildGuard judge. She also created detailed logging for attacker/victim inputs, rewards, judge labels, policy metadata, and trajectory inspection. This environment exposed the state, action, history, reward, and logging hooks used by the PPO implementation, while the PPO algorithm itself was developed separately by the team. Abi ran the main experimental suite, including PPO training, checkpoint sweeps, mutator-scale ablations, reward-signal probes, and all co-evolution ablations. For co-evolution, she implemented Modal orchestration, persistent trajectory banks, LoRA-based victim and attacker updates, cumulative training data, adapter verification, resume logic, and parallel rollout collection on A100 GPUs.

- **Nubia Correa:** Implemented TrailBlazer PPO checkpointing, Modal training and evaluation workflows, policy metadata logging, and attention-based history integration. Designed and ran the history ablation experiments at both 3-turn and 5-turn horizons with all three history entry points disabled for a clean ablation. Conducted RandomPolicy versus TrailBlazer evaluation experiments. Designed the framework pipeline diagram used in the paper and poster.
- **Annie Villalta:** Experimented with initial PPO training runs, extended them to longer training runs. During evaluation, I found that using Qwen as the judge produced noisy and inconsistent reward signals, so we switched to WildGuard for more reliable safety/reward evaluation. Designed and implemented ablation experiments across different interaction horizons: 1, 3, 5, 8, and 10 turns, in order to compare short-horizon versus long-horizon reward behavior.

Changes from Proposal We refined the implementation to emphasize lightweight LoRA-based adaptation and smaller open-source models to make iterative co-evolution experiments computationally feasible, while preserving the central attacker-victim dynamics described in the original proposal.

References

- Roman Belaire, Arunesh Sinha, and Pradeep Varakantham. 2025. Automatic LLM Red Teaming. arXiv:2508.04451 [cs.LG] <https://arxiv.org/abs/2508.04451>
- Sung-Hoon Yoon, Ruizhi Qian, Minda Zhao, Weiyue Li, and Mengyu Wang. 2026. TrailBlazer: History-Guided Reinforcement Learning for Black-Box LLM Jailbreaking. arXiv:2602.06440 [cs.CL] <https://arxiv.org/abs/2602.06440>
- Taeyoung Yun, Pierre-Luc St-Charles, Jinkyoo Park, Yoshua Bengio, and Minsu Kim. 2025. Active Attacks: Red-teaming LLMs via Adaptive Environments. arXiv:2509.21947 [cs.LG] <https://arxiv.org/abs/2509.21947>

A Implementation Details

A.1 PPO Policy Architecture

The attacker policy, `TrailBlazerPolicy`, cross-attends the current query/template state to a window of previous turn vectors and outputs both a discrete mutation-action distribution and a scalar value estimate. Text features are embedded with `sentence-transformers/all-MiniLM-L6-v2` using 384-dimensional L2-normalized embeddings with `max_length=256`. Each history turn vector concatenates a text embedding with structured features: action one-hot, reward, recency score, refusal flag, normalized response length, and judge-unsafe flag.

Trainable query, key, and value projections map these inputs to hidden size 256. The attended history context is concatenated with the projected current state, producing a 512-dimensional representation used by two-layer ReLU MLPs for the policy and value heads. The policy selects among five RLBreaker-style mutation actions: `rephrase`, `crossover`, `generate_similar`, `shorten`, and `expand`. The learned attention weights are also passed to the mutator prompt so that higher-weight prior turns are emphasized during template mutation.

A.2 PPO Training Hyperparameters

Table 11: PPO hyperparameters used for the main fixed-victim training runs.

Hyperparameter	Value
Number of epochs	100
Episodes per epoch	24
Max turns per episode	5
Learning rate	3×10^{-4}
Clip epsilon	0.2
Value loss coefficient	0.5
Entropy coefficient	0.01
Discount factor (γ)	0.99
GAE λ	0.95
Optimizer	Adam
GPU	A100-80GB

Each PPO epoch collects one rollout batch, computes GAE advantages, and performs one PPO gradient update. Checkpoints are saved every epoch and swept on held-out validation prompts to select the best policy. For co-evolution experiments, we start from the selected fixed-victim checkpoint and continue PPO training in shorter cycle-based phases.

A.3 Datasets

We draw harmful seed prompts from AdvBench, HarmBench, and a legacy seed set. AdvBench is used for PPO training and reward-probe experiments, while HarmBench validation and test prompts are used for checkpoint selection and held-out evaluation. After deduplication and merging, the final splits contain 2,503 training prompts, 325 validation prompts, and 564 test prompts. Seed prompts are inserted into attack templates through a protected `{REQUEST}` placeholder, so the mutator edits jailbreak templates rather than directly rewriting harmful requests.

A.4 State Features and History

The PPO state uses `history_window=4`, so up to four prior turns are included in the policy features. Separately, `victim_history_turns=3` controls how many prior user-assistant exchanges are prepended to the victim prompt, and `max_history_chars=4000` bounds the history shown to the mutator. In history-ON runs we use these default values. In history-OFF ablations, all three history channels are disabled.

A.5 Reward Backends

Our main reward backend is WildGuard, `allenai/wildguard`. WildGuard receives the rendered attacker prompt and victim response, then classifies whether the response is harmful. Responses classified as unsafe receive reward 1.0, and all other responses receive reward 0.0. We also implement an AdvBench reference-similarity backend inspired by RLBreker: victim responses and paired harmful reference completions are embedded with `BAAI/bge-large-en-v1.5`, and cosine similarity is used as a raw dense reward.

A.6 Parallel Rollout Collection

PPO rollouts are batched to better utilize the A100. Within each epoch, episodes advance one turn at a time in lockstep. At each turn, batched vLLM calls generate mutator outputs, victim responses, and WildGuard scores across the active episode batch. The fixed-victim PPO runs use 24 episodes per epoch. Co-evolution runs use 16 episodes per PPO epoch. To speed up collection, we run 8 episodes concurrently, so each PPO epoch is collected in two batched rollout chunks.

During inference-heavy runs, three vLLM models are active: the mutator, the victim, and the WildGuard judge. The default GPU memory utilization settings are for the mutator (`gpu_memory_utilization=0.2`), the victim (`gpu_memory_utilization=0.3`, LoRA-

enabled), and the WildGuard judge (`gpu_memory_utilization=0.35`). Checkpoints, rollout logs, trajectory banks, LoRA adapters, and summaries are persisted to the Modal volume `cs224r-redteam-rl-data`.

A.7 Co-evolution Schedule

Victim-only co-evolution	PPO rollouts → victim LoRA update
Attacker-only co-evolution	PPO rollouts → attacker LoRA update
Three-way co-evolution	PPO rollouts → attacker LoRA update → victim LoRA update

Figure 4: Co-evolution update schedules. All variants use the same PPO rollout budget; they differ only in which LoRA adaptation phases are applied after each cycle.

Co-evolution is organized into cycles. Each cycle begins with a PPO rollout/update phase, followed by optional attacker LoRA training and optional victim LoRA training. The output policy and adapters from one cycle become the inputs to the next cycle. This lets us instantiate several ablations from the same code path: PPO only, PPO with evolving victim, PPO with evolving attacker, and full three-way co-evolution.

For the main co-evolution experiments, we start from the selected Qwen2.5-1.5B fixed-victim PPO checkpoint at epoch 89. Each co-evolution run uses 4 cycles. Within each cycle, PPO trains for 15 epochs, with 16 episodes per epoch, 5 turns per episode, and rollout collection parallelized in batches of 8 episodes at a time. Thus, each co-evolution run performs 60 PPO epochs total, with 240 rollout episodes per cycle and 1,200 victim queries per cycle before the optional LoRA update phases. For the 4 cycle coevolution cycle there were 4,800 victim queries. The three-way co-evolution run uses the same PPO rollout budget, but each cycle performs both adaptation phases: the attacker mutator is updated with LoRA on successful template mutations, and the victim is updated with LoRA on successful attacks before the next PPO cycle begins. PPO keeps the same learning rate, clipping, entropy, value, discount, and GAE settings as the fixed-victim PPO runs.

Each co-evolution run uses 4 cycles. Within each cycle, PPO trains for 15 epochs, with 16 episodes per epoch, 5 turns per episode, and rollout collection parallelized in batches of 8 episodes at a time. This gives 240 rollout episodes and up to 1,200 victim queries per cycle, or 960 rollout episodes and up to 4,800 victim queries across the full 4-cycle run, before accounting for the LoRA update phases.

A.8 LoRA Adapter Training

Both victim and attacker evolution use LoRA adapters on the same base model, Qwen/Qwen2.5-1.5B-Instruct. Unless otherwise specified, adapters are trained for 100 gradient steps with learning rate 3×10^{-5} , weight decay 0, max gradient norm 1.0, bf16 weights, LoRA rank $r = 32$, LoRA alpha 16, dropout 0, and target modules `q_proj`, `k_proj`, `v_proj`, and `o_proj`.

Victim LoRA training converts successful attacks into supervised refusal examples. The instruction is the rendered attack prompt, and the target response is a fixed safe refusal template. In our co-evolution runs, victim training uses cumulative successful attacks from all prior cycles. For memory stability, the victim training override uses batch size 1, gradient accumulation 8, and max sequence length 768.

Attacker LoRA training converts successful mutator calls into supervised template-editing examples. The input is the full mutator prompt, including the selected action and template history, and the target is the successful attacker message/template. We support both current-cycle and cumulative attacker training; the final attacker-evolution and three-way co-evolution runs use cumulative attacker examples so that later cycles can continue learning from sparse successful mutations. Attacker training uses batch size 8, gradient accumulation 4, and max sequence length 2048.