MURPHY: Feedback-Aware GRPO with Retrospective Credit Assignment for Multi-Turn Code Generation

Chanakya Ekbote*
cekbote@mit.edu

Vijay Lingam*
vlingam@amazon.com

Sujay Sanghavi

Behrooz Omidvar-Tehrani

Jun Huan

Anoop Deoras

Stefano Soatto

Massachusetts Institute of Technology AWS AI Amazon

Abstract

Reinforcement Learning with Verifiable Rewards (RLVR) has become a standard recipe for post-training LLMs on reasoning tasks, with Group Relative Policy Optimization (GRPO) emerging as a leading approach. However, GRPO and its variants are inherently single-turn: they optimize from terminal rewards on isolated prompt-response pairs, leaving them poorly suited to agentic settings where models must iteratively refine solutions in response to environmental feedback. We introduce MURPHY, a multi-turn extension of GRPO for self-correcting code generation. MURPHY constructs feedback-conditioned rollout trees in which failed candidate solutions are paired with executor feedback and expanded into subsequent turns, and propagates rewards backward through the tree so that later successful refinements credit earlier attempts that surfaced informative feedback. We study two propagation strategies, Max Reward (MARS) and Mean Reward (MERS), and introduce post-rollout pruning mechanisms that reduce multi-turn optimization cost. Across three code generation benchmarks (HumanEval, MBPP, LiveCodeBench-v6) and two model families (Qwen3-1.7B/4B, OLMo-2-7B), MURPHY delivers up to **6%** absolute pass@1 gains over the strongest prior multi-turn execution-feedback methods. Gains are largest on the Medium/Hard subset (**+4.38/+4.20** at Iter-5), where iterative self-correction matters more.

1 Introduction

> *"The road to wisdom? Well, it's plain and simple to express: err and err and err again, but less and less and less."* —*Piet Hein*

Large language models (LLMs) are increasingly deployed as code-generation agents that interact with their environment rather than producing a single static response [30, 24]. In a typical execution-feedback scaffold [21, 38, 15], the model proposes code, the code is executed against unit tests, and failures are returned as error messages, stack traces, or failing test cases. The model is then re-prompted with the original task, its previous attempt, and the feedback, and this process repeats until the solution is accepted or a turn budget is exhausted. Such scaffolds substantially improve

*Equal contribution and corresponding author(s). Chanakya Ekbote was an intern at AWS AI during this work.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

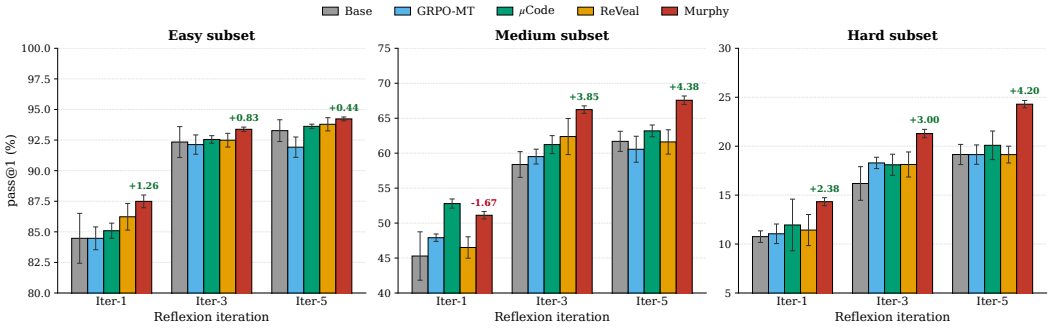


Figure 1: **Self-correction performance on LiveCodeBench-v6.** Qwen3-4B with MURPHY vs. four execution-feedback baselines (Base, GRPO-MT, μCODE, REVEAL) on the Easy, Medium, and Hard subsets. Base, GRPO-MT, μCODE, and MURPHY are evaluated under the Reflexion framework [21]; REVEAL is evaluated using its official generation-verification scaffold, which uses a method-specific prompt and parser. All methods are compared at 1, 3, and 5 inference turns. Bars show the mean and stdev across 3 runs. MURPHYshows large gains on the Medium/Hard subset (**+4.38/+4.20** at Iter-5), where self-correction matters more.

inference-time self-correction, but they leave the underlying model unchanged: feedback is consumed as test-time context only [21, 19].

Recent works attempt to close this gap by training models to use execution feedback. RLEF [8] optimizes multi-turn code generation with PPO [17] against execution-based rewards. μCODE [9] recasts multi-turn correction as expert-iteration imitation learning with a learned verifier for relabeling and selection. REVEAL [12] interleaves generation and verification within a single policy, using turn-aware optimization to improve self-verification. These methods establish the value of execution feedback, but each relies on PPO-style value estimation, learned verifiers, or explicit verification policies. They leave open a simpler question: **can the standard GRPO** [20] **post-training recipe itself be made feedback-aware in multi-turn environments?**

A natural baseline is GRPO-MT, which extends GRPO by treating each feedback-conditioned interaction as a trajectory. The model samples candidates for the original prompt; passing rollouts terminate, while failed ones are paired with executor feedback and expanded for additional turns until solved or the turn budget is exhausted. Each terminated rollout's final execution reward serves as its trajectory-level outcome for the GRPO advantage, broadcast across policy generated tokens along that rollout while executor feedback and prior-solution context are masked from the loss. Although GRPO-MT exposes the model to execution feedback, its credit assignment is coarse: all generations along a terminated trajectory inherit the same terminal signal, regardless of whether an intermediate attempt produced useful feedback, a misleading direction, or contributed little to the eventual fix. This matters in code generation, where failed attempts are not uniformly bad: an incorrect solution may reveal the precise failing case or runtime behavior that enables a later refinement. GRPO-MT can reinforce successful trajectories but cannot distinguish informative failures from uninformative ones when they share a terminal outcome.

Prior inference-time agent frameworks [21, 38, 15] show that previous failed attempts and environment feedback provide useful signals for self-correction. In code generation, a failed program may expose an edge case, runtime error, or test failure that enables a later refinement. Thus, failed attempts should not be treated as uniformly negative; their value depends on whether the feedback they induce supports future correction.

MURPHY turns this observation into a training-time objective for multi-turn GRPO. It constructs feedback-conditioned rollout trees in which successful candidates terminate and failed candidates are expanded by re-prompting on the original problem, prior output, and executor feedback. Once the rollout is complete, MURPHY propagates rewards backward through the tree, allowing later successful refinements to credit earlier attempts. We instantiate this idea with Max Reward (MARS), which backs up the best descendant outcome, and Mean Reward (MERS), which backs up a discounted average future reward, then apply a local GRPO update using the resulting credit-adjusted rewards. This design isolates the algorithmic question from confounding system choices. Unlike RLEF, MURPHY requires

no PPO critic; unlike μCODE, no learned verifier or imitation reduction; unlike REVEAL, no separate verification behavior or test synthesis. MURPHY is a minimal, architecture-preserving extension of GRPO to feedback-conditioned, multi-turn code generation.

Main Contributions.

1. We introduce MURPHY, a multi-turn GRPO algorithm that trains code LLMs to use execution feedback through feedback-conditioned rollouts and structured temporal credit assignment. (Sec. 3)
2. We propose MARS and MERS, two reward-propagation strategies that assign credit from later refinements to earlier attempts in the rollout tree.
3. We design pruning strategies that make multi-turn GRPO training tractable by retaining informative trajectories while reducing optimization cost. (Subsec. 3.1)
4. We evaluate MURPHY on three code-generation benchmarks across two model families (OLMo, Qwen) and sizes (1.7B–7B), achieving up to **6%** absolute pass@1 gains over prior multi-turn execution-feedback methods. (Sec. 4)

2 Background: GRPO

Group Relative Policy Optimization (GRPO; [20]) is a reinforcement-learning objective for LLM fine-tuning that avoids learning a separate value function. For each input prompt q, the old policy $\pi_{\theta_{\text{old}}}$ samples a group of G candidate responses, $\mathcal{C}(q) = \{u_1, \ldots, u_G\}$. Each response $u \in \mathcal{C}(q)$ receives a scalar reward $r(u)$. GRPO computes advantages by normalizing rewards within the response group:

$$\hat{A}_q^{\text{GRPO}}(u) = \frac{r(u) - \text{mean}_{w \in \mathcal{C}(q)} r(w)}{\text{std}_{w \in \mathcal{C}(q)} r(w)}, \qquad u \in \mathcal{C}(q) \tag{1}$$

This group-relative baseline replaces the learned critic used in PPO. Let u_t denote the t-th token of response u, and let $u_{<t}$ be its prefix. The token-level importance ratio is $R_\theta(q, u, t) = \frac{\pi_\theta(u_t | q, u_{<t})}{\pi_{\theta_{\text{old}}}(u_t | q, u_{<t})}$. The GRPO clipped objective for a response group is then

$$\mathcal{J}_{\text{GRPO}}\left(\theta; q, \mathcal{C}(q), \hat{A}_q\right) = \mathbb{E}_{u \in \mathcal{C}(q)}\left[\frac{1}{|u|}\sum_{t=1}^{|u|} \min\Big(R_\theta(q,u,t)\hat{A}_q(u), \text{clip}(R_\theta(q,u,t), 1-\epsilon, 1+\epsilon)\hat{A}_q(u)\Big)\right] - \beta\, D_{\text{KL}}(\pi_\theta \,\|\, \pi_{\text{ref}}) \tag{2}$$

where π_{ref} is a fixed reference policy and β controls the KL penalty. In standard single-turn GRPO, $\hat{A}_q = \hat{A}_q^{\text{GRPO}}$ is computed from the immediate rewards of responses sampled for the same prompt. MURPHY keeps this group-relative update structure but changes how the rewards used in the advantage are assigned in a multi-turn rollout tree as described in Sec. 3.

3 Proposed Method: MURPHY

MURPHY extends GRPO to feedback-rich, multi-turn code generation. Standard GRPO optimizes one response group per prompt using rewards from that same generation step. This is insufficient for agentic code generation, where the model may execute code, observe failures, and refine its solution over multiple attempts. In this setting, an initially incorrect solution can still be valuable if it exposes useful executor feedback, such as a failing edge case or runtime error, that enables a later correction. MURPHY addresses this mismatch by constructing feedback-conditioned rollout trees and assigning credit retrospectively from later successful refinements to earlier attempts.

Feedback-conditioned rollout tree. For each input prompt, MURPHY first samples a group of candidate programs. Each candidate is executed against the task test suite, producing two signals: a scalar reward, such as the proportion of tests passed, and qualitative executor feedback, such as failing test cases, stack traces, compiler errors, or timeout messages. Candidates that achieve the

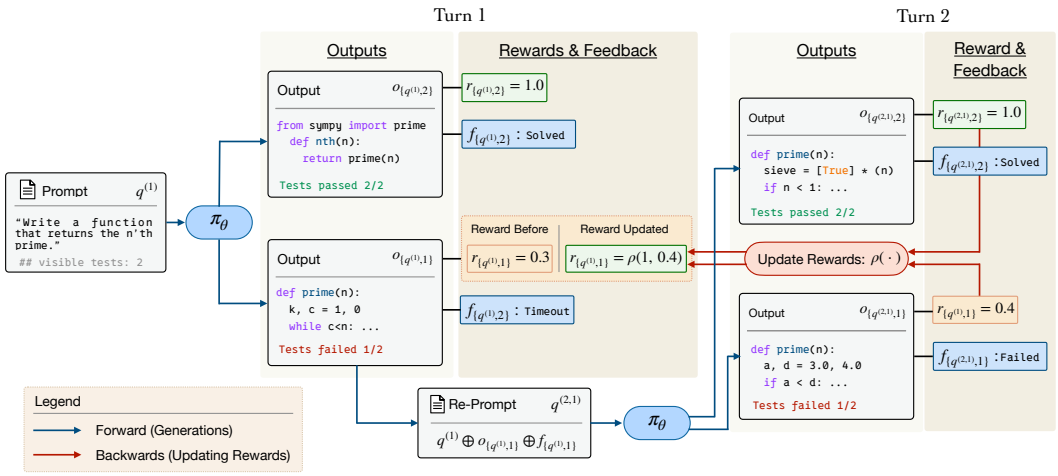


Figure 2: **Overview of MURPHY.** Given an input prompt $q_{(1)}$, the model samples multiple candidate solutions. Each output is executed to obtain a scalar reward r and executor feedback f. Solved outputs terminate, while failed outputs are converted into feedback-conditioned prompts by concatenating the original prompt, the failed output, and its feedback. The model then samples refinements for each failed output. After the rollout tree is generated, rewards from later turns are propagated backward through a credit-assignment rule $\rho(\cdot)$, allowing successful refinements to credit earlier failed attempts that produced useful feedback.

maximum reward are treated as solved and become terminal nodes: they are kept for optimization but are not expanded in later turns.

Failed candidates are expanded in the next turn. To do so, MURPHY forms a new prompt by concatenating the original task prompt, the failed output, and the corresponding executor feedback. The policy is then re-invoked on this feedback-conditioned prompt to generate a new group of candidate refinements. This expansion is local to each failed node: different failures produce different feedback-conditioned prompts and therefore induce different refinement groups. The process repeats for a fixed number of turns or until no failed nodes remain. The result is a rollout tree in which each node is a generated program and each edge corresponds to one refinement step conditioned on executor feedback from the parent node.

This tree structure preserves the causal relationship between an attempted solution, the feedback it produces, and the refinements that follow. In Fig. 2, the first turn produces two outputs. The solved output terminates immediately. The failed output receives feedback, is converted into a new prompt, and is expanded into two second-turn refinements. Thus, later outputs are not independent samples from the original prompt; they are feedback-conditioned descendants of specific earlier attempts.

Retrospective credit assignment. After the rollout tree is generated and all candidates are evaluated, MURPHY propagates rewards backward using a bottom-up recursion. Starting from the leaves, each node passes reward information to its parent; the parent is then updated using its own immediate execution reward and the adjusted rewards of its children. This differs from single-turn GRPO, where a response is credited only by its own reward within the original response group. In MURPHY, a failed attempt can receive positive credit if its feedback leads to a later successful refinement.

Credit-adjusted rewards. Let $\mathcal{T}$ denote the rollout tree for a prompt, where each node $v \in \mathcal{T}$ corresponds to a generated program. Each node has an immediate execution reward $r(v)$ and, if expanded, a child response group $\mathcal{C}(v)$ generated by re-prompting on its feedback. Solved nodes are terminal and have $\mathcal{C}(v) = \emptyset$. MURPHY computes credit-adjusted rewards $\tilde{r}(v)$ by a backward

recursion over $\mathcal{T}$. For leaf nodes, $\tilde{r}(v) = r(v)$. For rest nodes, we consider 2 credit-assignment rules:

$$\textsc{MARS}: \quad \tilde{r}(v) = \max\left(r(v), \max_{u \in \mathcal{C}(v)} \tilde{r}(u)\right), \tag{3}$$

$$\textsc{MERS}: \quad \tilde{r}(v) = \frac{r(v) + \gamma \cdot \text{mean}_{u \in \mathcal{C}(v)} \tilde{r}(u)}{Z(v)}, \tag{4}$$

where $\gamma \in [0, 1]$ is a discount factor and $Z(v)$ is a normalization term that counts the reward components included in the backup. This keeps adjusted rewards comparable across nodes at different depths. In our two-turn setting, $Z(v) = 2$ for unsolved internal nodes, corresponding to the node's immediate reward and the propagated reward from its children, while terminal nodes retain $Z(v) = 1$ and $\tilde{r}(v) = r(v)$. MARS performs an optimistic backup: it propagates the best achievable refinement outcome upward through the tree, similar in spirit to max-backup variants of Monte Carlo Tree Search [13]. Thus, if one refinement of a failed first-turn solution solves the task, MARS credits that success back to the first-turn node. MERS instead propagates an average future-refinement value, analogous to a Bellman-style backup over sampled refinement nodes. Unlike value-based RL, however, MURPHY performs this backup directly over generated refinements rather than learning a separate value function.

Node-wise GRPO objective. After credit assignment, MURPHY applies GRPO locally to each response group in the rollout tree. Let $\mathcal{Q}(\mathcal{T})$ denote the set of prompts and feedback-conditioned prompts expanded in the tree. For any $q \in \mathcal{Q}(\mathcal{T})$, let $\mathcal{C}(q)$ be its response group. Instead of computing advantages from immediate rewards as in Eq. 1, MURPHY normalizes credit-adjusted rewards within the group:

$$\hat{A}_q^{\textsc{MURPHY}}(u) = \frac{\tilde{r}(u) - \text{mean}_{w \in \mathcal{C}(q)} \tilde{r}(w)}{\text{std}_{w \in \mathcal{C}(q)} \tilde{r}(w)}, \qquad u \in \mathcal{C}(q). \tag{5}$$

The MURPHYobjective is the sum of GRPO losses over the expanded response groups:

$$\mathcal{J}_{\textsc{MURPHY}}(\theta) = \mathbb{E}_{\mathcal{T} \sim \pi_{\theta_{\text{old}}}} \left[\sum_{q \in \mathcal{Q}(\mathcal{T})} \mathcal{J}_{\text{GRPO}}\left(\theta; q, \mathcal{C}(q), \hat{A}_q^{\textsc{MURPHY}}\right) \right] \tag{6}$$

Thus, MURPHY does not introduce a critic or a new policy-gradient estimator. It applies the same GRPO update from Eq. 2 at each expanded response group in the rollout tree, while replacing immediate rewards with retrospectively adjusted rewards. The policy update remains token-level, but the scalar learning signal attached to each generation now reflects both its immediate execution result and its contribution to later feedback-conditioned refinements. We provide the full rollout-tree indexing, recursive credit-assignment details, and implementation details in App. C.

3.1 Pruning Strategies in MURPHY

The feedback-conditioned rollout tree in MURPHY can grow quickly with the number of turns. By default, MURPHY uses a fixed generation budget G at every turn. If most generations fail and continue to be expanded, the number of generated programs can grow as G^S for a turn budget S, in the worst case. This increases memory use and slows optimization, since each retained sequence may contribute to advantage computation and gradient updates. To make multi-turn training tractable, we introduce pruning strategies that retain a smaller subtree before retrospective credit assignment.

Pruning is applied after rollouts are generated and rewards are computed, but before rewards are propagated backward. This order is important: the rollout procedure still collects feedback-conditioned generations and their rewards, but only the retained subtree contributes to credit assignment and optimization. Let $\mathcal{T}' \subseteq \mathcal{T}$ denote the retained subtree after pruning ($\mathcal{T}' = \text{Prune}(\mathcal{T})$). Pruning changes both the children used in the backward credit recursion and the response groups included in the node-wise GRPO objective:

$$\mathcal{J}_{\textsc{MURPHY}}^{\text{pruned}}(\theta) = \mathbb{E}_{\mathcal{T} \sim \pi_{\theta_{\text{old}}}} \left[\sum_{q \in \mathcal{Q}(\mathcal{T}')} \mathcal{J}_{\text{GRPO}}\left(\theta; q, \mathcal{C}_{\mathcal{T}'}(q), \hat{A}_q^{\textsc{MURPHY}}\right) \right], \tag{7}$$

where $\mathcal{C}_{\mathcal{T}'}(q)$ denotes the retained response group for prompt q in the pruned tree. In this sense, pruning is a compute-aware filter on the rollout tree: it does not change the rollout mechanism or the GRPO update, but determines which generated trajectories are retained for reward propagation and gradient updates.

We consider two pruning strategies with different budget granularities. INTRAP uses a *within-group trajectory budget*, retaining a fixed number of trajectories inside each response group. INTERP uses an *across-group budget*, retaining a fixed number of refinement groups among those induced by failed parent nodes. We use $\mathrm{TopB}^{b}_{x\in\mathcal{X}}(s(x))$ to denote the subset of b elements from $\mathcal{X}$ with the largest scores $s(x)$.

Intra-Group Pruning (INTRAP). INTRAP prunes individual trajectories within each response group. Consider a prompt or feedback-conditioned prompt q with a group of generated children $\mathcal{C}(q)$. Given a within-group trajectory budget b_{traj}, INTRAP retains a subset $\mathcal{C}'(q) \subseteq \mathcal{C}(q)$ with $|\mathcal{C}'(q)| = b_{\mathrm{traj}}$ and discards the remaining children together with their descendants. The retained trajectories are selected according to their contribution to within-group reward variance: $\mathcal{C}'(q) = \mathrm{TopB}^{b_{\mathrm{traj}}}_{u\in\mathcal{C}(q)}\left(\Delta_{\mathrm{Var}}(u;\mathcal{C}(q))\right)$, where $\Delta_{\mathrm{Var}}(u;\mathcal{C}(q))$ measures how much trajectory u contributes to the reward variance of the group. This strategy is inspired by [27], who show that retaining high-variance trajectories within a response group can reduce optimization cost while preserving the learning signal needed for group-relative policy optimization. We extend this idea to the multi-turn setting by applying it recursively over the rollout tree. Since GRPO computes advantages from relative reward differences within a group, preserving reward diversity helps retain informative comparisons while reducing the number of sequences used for optimization. If retained samples have nearly identical rewards, the normalized advantages provide little useful preference signal; INTRAP therefore keeps trajectories that maintain reward contrast within each group.

Inter-Group Pruning (INTERP). INTERP prunes at the level of entire refinement groups. Each failed node at turn s creates a feedback-conditioned prompt at turn $s+1$, and that prompt produces a group of child refinements. Instead of selecting individual trajectories within every group, INTERP selects which refinement groups to retain. The intuition is that not all feedback contexts are equally useful for optimization. Some failed attempts lead to refinement groups where all children behave similarly: they may all solve the task, or they may all fail. Such groups provide limited relative learning signal because there is little contrast among candidates generated from the same feedback-conditioned prompt. Other failed attempts lead to mixed refinement groups, where some children improve substantially while others still fail. These groups are more informative because they reveal which refinements should be preferred under the same feedback context. Based on this intuition, INTERP ranks refinement groups by their within-group reward variance. Let $\mathcal{G}_s$ denote the set of child groups induced by failed nodes at turn s. Given an across-group budget b_{grp}, INTERP retains the b_{grp} groups with the largest within-group reward variance: $\mathcal{G}'_s = \mathrm{TopB}^{b_{\mathrm{grp}}}_{g\in\mathcal{G}_s}\left(\mathrm{Var}\big(\{r(u) : u \in g\}\big)\right)$. The remaining groups, along with all of their descendants, are discarded.

This pruning rule is aligned with the group-relative nature of GRPO. Since advantages are computed by comparing rewards within a group, high-variance groups provide stronger preference information than groups whose rewards are nearly uniform. In contrast to INTRAP, which keeps informative trajectories within each group, INTERP allocates optimization budget across feedback-conditioned prompts by keeping the refinement groups that appear most informative for learning. In App. H, we provide a discussion on the practical cost of rollout tree generation.

4 Experiments

In this section, we first provide an overview of metrics and evaluation protocol used in our experiments, followed by detailed results, including ablation studies. We provide implementation details including inference hyper-parameters in App. F.

Metrics and Evaluation Protocol. Reflexion [21] is a widely adopted purely inference-time multi-turn framework: the model parameters are not updated, but the model iteratively refines its solution by conditioning on prior attempts, execution feedback from visible test cases, and self-generated reflections (see App. E for details). To assess refinement and self-correction, we integrate all models into the same Reflexion scaffold and report *pass@1* under K-iteration evaluation.

We denote these settings as Iter-K: Iter-1 corresponds to standard input-output prompting without feedback, while at iteration $k > 1$ the model revises its solution using a sliding-window history consisting of the previous output and its corresponding visible-test feedback. Unless otherwise specified, we use Iter-3 as the default multi-turn evaluation setting. The agent terminates early once all visible tests pass or when the maximum iteration budget is reached. Final solutions are evaluated on hidden test cases, and the resulting *pass@1* is reported. Each experiment is repeated three times, and we report the mean and standard deviation of *pass@1* across runs.

HumanEval and MBPP. We evaluate OLMo-2-1124-7B-Instruct [16] and Qwen3 (1.7B, 4B) [29], each fine-tuned on 1,000 samples randomly drawn from KodCode [28] and evaluated on HumanEval [5] and MBPP [2] using the evaluation protocol described above. KodCode is chosen for its minimal overlap with evaluation benchmarks[2]. We report *pass@1* under single-iteration (Iter-1) and three-iteration (Iter-3) Reflexion settings in Fig. 3; in all experiments, MURPHY uses MARS reward propagation.

Iter-1 evaluates one-shot generation and does not exercise feedback-conditioned refinement, so we treat it primarily as a sanity check that multi-turn training does not substantially degrade direct generation. In this setting, MURPHY is competitive with the strongest baseline with one small regression on Qwen3-1.7B MBPP (**-1.27 pp**). The primary evaluation for self-correction is Iter-3, where models can condition on execution feedback and revise their previous attempts. Under this setting, MURPHY outperforms all baselines, with gains up to **+6.20 pp** over the strongest comparator. The largest improvements occur on OLMo-2-1124-7B-Instruct (**+6.20 pp** on HumanEval and **+6.07 pp** on MBPP), suggesting that weaker models benefit more from structured multi-turn credit assignment. Overall, these results indicate that retrospective reward propagation primarily improves feedback-conditioned refinement while largely preserving one-shot generation performance.

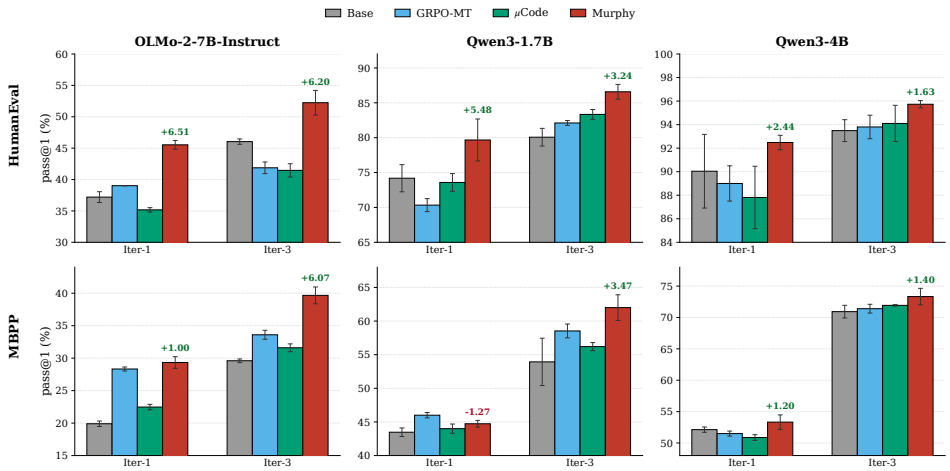


Figure 3: **HumanEval and MBPP pass@1 across Reflexion iterations for three base models.** Rows correspond to benchmarks (HumanEval, MBPP) and columns to base models (OLMo-2-1124-7B-Instruct, Qwen3-1.7B, Qwen3-4B). Within each cell, we compare MURPHY (ours) against the Base model, the multi-turn baseline GRPO-MT, and the execution-feedback method μCode [9]. Annotations above MURPHY's bars report the absolute percentage-point difference between MURPHY and the strongest non-MURPHY baseline within the same benchmark, model, and Reflexion iteration (green: improvement, red: regression). MURPHY offers superior performance in Iter-3 settings, where feedback-conditioned self-correction is directly evaluated. Detailed results in Tab. 4.

LiveCodeBench-v6. We additionally include REVEAL [12], a concurrent multi-turn RL framework that evolves code generation through self-verification and tool-based evaluation, in our most rigorous setting: the contamination-resistant LiveCodeBench-v6 benchmark with Qwen3-4B (Fig. 1). Base, GRPO-MT, μCODE, and MURPHY are evaluated under the same Reflexion scaffold. For REVEAL,

[2]Refer to Section 3.2 of KodCode [28] for details on contamination analysis.

Table 1: MARS vs. MERS reward propagation. MARS is highlighted. Δ_3 is relative to MERS ($\gamma = 1$) Iter-3 within each model block.

Model / Strategy	HumanEval			MBPP		
	Iter-1	Iter-3	Δ_3	Iter-1	Iter-3	Δ_3
Qwen3-1.7B						
MARS	**79.67 ± 3.01**	**86.58 ± 1.06**	+1.01	44.73 ± 0.50	**62.00 ± 1.91**	+1.07
MERS ($\gamma = 0.9$)	78.66 ± 2.11	84.76 ± 1.22	–0.81	**46.53 ± 0.81**	60.53 ± 1.79	–0.40
MERS ($\gamma = 1$)	78.46 ± 0.70	85.57 ± 0.35	–	45.07 ± 1.10	60.93 ± 1.29	–
OLMo-2-1124-7B-Instruct						
MARS	**45.53 ± 0.70**	**52.24 ± 1.96**	+10.37	**29.33 ± 0.90**	**39.67 ± 1.29**	+5.27
MERS ($\gamma = 1$)	37.40 ± 0.35	41.87 ± 1.27	–	27.87 ± 0.12	34.40 ± 2.40	–

we instead use its official repository, hyperparameters, and native multi-turn generation-verification scaffold with 3 training turns. This gives REVEAL its intended evaluation setting, since its inference procedure relies on a method-specific generation prompt, output format, and parser. Although the prompt and parser differ from Reflexion, the underlying scaffolds are conceptually similar: both condition on prior incorrect responses and environment feedback over multiple turns to refine subsequent attempts. All methods are compared at 1, 3, and 5 inference turns. MURPHY outperforms both μCODE and REVEAL across all difficulty subsets, with the largest gain (+4.20 pp) on the Hard subset at Iter-5, where self-correction matters most. Detailed results, including scaffold robustness analyses, are available in F.4.1 (MURPHY's gains remain consistent across multiple scaffolds).

Scope of REVEAL comparison. REVEAL's co-trained self-verifier substantially increases per-experiment training cost: ~23 hours end-to-end for Qwen3-4B versus ~11 hours for MURPHY on matched hardware (8×H100s). We therefore include ReVeal as a baseline only in our most rigorous setting (LCB-v6 with Qwen3-4B, Fig. 1), where contamination resistance and difficulty stratification make the comparison most informative. Extending REVEAL across all model–benchmark cells in Fig. 3 was prohibitive under our compute budget, mirroring our handling of RLEF [8].

Additional Experiments. Further analysis is provided in the appendices: effect of training data size (App.F.4.2), and training beyond two turns (App.F.4.3). App. D confirms that MURPHY's gains generalize across different iterative scaffolds.

4.1 Reward Propagation Ablation: MARS vs. MERS

We ablate the two reward propagation strategies, MARS and MERS, introduced in Subsec. 3.1, training Qwen3-1.7B and OLMo-2-1124-7B-Instruct on 1K KodCode subset dataset under each strategy (Tab. 1). MARS offers competitive or superior performance over MERS in iter-3 setting independent of the discount factor γ. The gap stems from how each strategy handles non-binary rewards: MERS averages over child rewards, diluting the signal when few refinements score highly, whereas MARS propagates the strongest outcome, letting rare high-reward trajectories dominate the update. This advantage is most pronounced under sparse rewards and diminishes as rewards become binary.

4.2 Pruning Strategies Ablation: INTRAP vs. INTERP

We ablate the two pruning strategies, Intra-Group (INTRAP) and Inter-Group (INTERP), introduced in Subsec. 3.1, training Qwen3-1.7B on 1,000 KodCode samples under each strategy (Tab. 2). Both strategies reduce the number of retained trajectories per rollout tree (per prompt) from 72 to roughly half, lowering the per-step optimization cost without altering the rollout procedure itself. The two strategies differ in which trajectories they retain, and this leads to different multi-turn behaviors: INTRAP improves single-turn performance (Iter-1) on both benchmarks but regresses by **2–2.3 pp** at Iter-3, whereas INTERP preserves multi-turn performance within **0.4–0.8 pp** of unpruned MURPHY. INTERP therefore offers the better cost-quality trade-off, retaining the multi-turn self-correction signal that pruning could otherwise discard.

Efficiency gains. To localize the speedup, we profile per-step timing averaged across an end-to-end INTERP run against unpruned MURPHY under matched configurations (Qwen3-1.7B, 8×H100

Table 2: Pruned variants of MURPHY on Qwen3-1.7B. All variants generate 72 rollout trajectories per query before pruning; the *Retained* column reports the number of trajectories kept after pruning per prompt, which determines how many sequence-level terms enter the GRPO loss. Δ_3 compares **Iter-3** against unpruned MURPHY (MARS) **Iter-3**; red indicates regression.

Model & Strategy	Retained	HumanEval (%)			MBPP (%)		
		Iter-1	Iter-3	Δ_3	Iter-1	Iter-3	Δ_3
MURPHY (MARS)	72	79.67 ± 3.01	**86.58 ± 1.06**	–	44.73 ± 0.50	**62.00 ± 1.91**	–
MURPHY (MARS) – INTRAP	36	**82.34 ± 1.03**	84.28 ± 2.01	–2.30	**45.73 ± 0.50**	59.87 ± 1.70	–2.13
MURPHY (MARS) – INTERP	40	77.43 ± 2.20	86.17 ± 0.70	–0.41	44.53 ± 0.90	61.20 ± 1.39	–0.80

GPUs, 72 rollouts per prompt: 8 in turn-1 and up to 64 in turn-2); see Tab. 8. Generation (vLLM) and reward computation (code execution) are essentially unchanged (-0.3% and -8.0% respectively), because pruning is applied *after* rollouts are collected. The savings concentrate entirely in the optimization phase, where pruning reduces the number of sequences entering gradient computation: optimization time drops by $\mathbf{74.1}\%$, yielding a $\mathbf{21.4}\%$ reduction in average per-step training time. This confirms that INTERP's computational benefit comes from avoiding gradient updates on uninformative trajectories, not from shortening rollouts, an important property, since aggressive rollout-time pruning would risk discarding feedback signal that later turns rely on.

We present additional experiments and analyses on feedback format and quality (Subsec. J.1), retrospective credit assignment (Subsec. J.2), training-time rollout depth (Subsec. J.3), failure-gated expansion and training cost (Subsec. J.4), scaffold-controlled baseline comparisons (Subsec. J.5), and generalization to Sudoku (Subsec. K.1) and repository-level software engineering (Subsec. K.2).

5 Related Work

LLM Agents for Software Development. Recent studies [10, 37] investigate LLM agents for code generation, bug fixing, and code migration. A central factor behind their progress is inference-time iterative frameworks [21, 15], which leverage execution feedback and self-reflection to refine candidate programs [30, 26]. While such methods enhance inference pipelines, they leave the base model unchanged. In contrast, our work improves the model itself through training-time optimization, strengthening the reasoning and self-correction abilities that agentic frameworks depend on.

RLVR for LLM Reasoning. GRPO [20] renewed interest in RL as an efficient alternative to PPO [17] for post-training LLMs on verifiable objectives, and follow-up variants [33, 31, 32, 36] improve stability, convergence, or shift optimization from token-level to sequence-level. These methods, however, remain single-turn. A separate line of work incorporates execution feedback during training but does so by introducing auxiliary machinery: μCODE [9] trains a separate verifier model that scores candidate solutions and relabels trajectories via expert iteration; RLEF [8] optimizes multi-turn rollouts with a PPO critic over execution-based rewards; and REVEAL [12] co-trains a self-verifier within the policy and uses turn-aware optimization to drive co-evolution of generation and verification. Concurrent work on multi-turn GRPO for tool-use agents [34] introduces turn-level rewards based on predefined interaction signals (e.g., tool-execution success). In code generation, execution can provide a scalar reward at each turn, but such intermediate rewards are often sparse or myopic: a failed program may receive low reward while exposing a stack trace, edge case, or failing test that enables a later correction. Thus, MURPHY uses retrospective tree-based propagation to assign credit according to downstream refinements induced by each feedback context. Concurrent to our work, TREEGRPO [7] also introduces tree-structured credit assignment for GRPO-style post-training, but in a different setting: visual generative models, where the tree branches within a single denoising trajectory and leaf advantages are propagated to denoising edges. In contrast, MURPHY constructs feedback-conditioned trees across multi-turn code-refinement attempts, propagates execution rewards before local group-relative normalization, and studies both optimistic (MARS) and expectation-like (MERS) backups. We provide a detailed comparison in App A.1. Across these lines of work, MURPHY differs by extending GRPO to multi-turn code generation through feedback-conditioned rollout trees and retrospective reward propagation, without a learned verifier, an auxiliary critic, or predefined dense turn-level rewards. Empirically, this minimal extension matches or outperforms μCODE and REVEAL across our benchmarks (Sec. 4). See App. A for an extended discussion.

6 Conclusion & Limitations

We introduced MURPHY, a multi-turn extension of GRPO that addresses the credit-assignment mismatch in feedback-conditioned code generation. MURPHY constructs feedback-conditioned rollout trees, then propagates rewards backward so that earlier failed attempts receive credit for the informative feedback they surface. Across three benchmarks and two model families, MURPHY delivers up to 6% absolute pass@1 gains over the strongest prior multi-turn execution-feedback methods, with the largest improvements on harder problems where iterative self-correction matters most. These results suggest that the standard GRPO recipe can be made feedback-aware through a structural change to credit assignment, without learned verifiers or auxiliary critics. **Limitations:** MURPHY's multi-turn rollouts increase training cost relative to GRPO. INTERP mitigates this cost, reducing average per-step training time, but multi-turn training remains more expensive due to rollout-tree expansion. In our experiments, we use vLLM for rollout generation; shared prefixes across parent and child nodes could further enable prefix caching and KV-cache reuse, which we leave to future work. Our evaluation focuses on code generation, where execution provides clean feedback. Extending MURPHY to noisier feedback signals, longer refinement horizons, broader agentic tasks, and adaptive turn or rollout budgets is a natural next step.

References

[1] A. Ahmadian, C. Cremer, M. Gallé, M. Fadaee, J. Kreutzer, O. Pietquin, A. Üstün, and S. Hooker. Back to basics: Revisiting REINFORCE-style optimization for learning from human feedback in LLMs. In L.-W. Ku, A. Martins, and V. Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12248–12267. Association for Computational Linguistics, Aug. 2024.

[2] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton. Program synthesis with large language models, 2021.

[3] T. Cai, Y. Li, Z. Geng, H. Peng, J. D. Lee, D. Chen, and T. Dao. Medusa: Simple llm inference acceleration framework with multiple decoding heads. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

[4] M. Chen, L. Sun, T. Li, H. Sun, Y. Zhou, C. Zhu, H. Wang, J. Z. Pan, W. Zhang, H. Chen, F. Yang, Z. Zhou, and W. Chen. Research: Learning to reason with search for llms via reinforcement learning, 2025.

[5] M. Chen, J. Tworek, and H. J. et. al. Evaluating large language models trained on code, 2021.

[6] DeepSeek-AI, D. Guo, and D. Y. et. al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025.

[7] Z. Ding and W. Ye. TreeGRPO: Tree-advantage GRPO for online RL post-training of diffusion models. In *The Fourteenth International Conference on Learning Representations*, 2026.

[8] J. Gehring, K. Zheng, J. Copet, V. Mella, T. Cohen, and G. Synnaeve. RLEF: Grounding code LLMs in execution feedback with reinforcement learning. In *Forty-second International Conference on Machine Learning*, 2025.

[9] A. K. Jain, G. Gonzalez-Pumariega, W. Chen, A. M. Rush, W. Zhao, and S. Choudhury. Multi-turn code generation through single-step rewards. In *Forty-second International Conference on Machine Learning*, 2025.

[10] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim. A survey on large language models for code generation. *ACM Trans. Softw. Eng. Methodol.*, July 2025.

[11] B. Jin, H. Zeng, Z. Yue, J. Yoon, S. O. Arik, D. Wang, H. Zamani, and J. Han. Search-r1: Training LLMs to reason and leverage search engines with reinforcement learning. In *Second Conference on Language Modeling*, 2025.

[12] Y. Jin, K. Xu, H. Li, X. Han, Y. Zhou, C. Li, and J. Bai. Reveal: Self-evolving code agents via reliable self-verification. In *The Fourteenth International Conference on Learning Representations*, 2026.

[13] P. Khandelwal, E. Liebman, S. Niekum, and P. Stone. On the analysis of complex backup strategies in monte carlo tree search. In *Proceedings of the 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1319–1328. PMLR, 2016.

[14] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica. Efficient memory management for large language model serving with pagedattention, 2023.

[15] V. Lingam, B. O. Tehrani, S. Sanghavi, G. Gupta, S. Ghosh, L. Liu, J. Huan, and A. Deoras. Enhancing language model agents using diversity of thoughts. In *The Thirteenth International Conference on Learning Representations*, 2025.

[16] T. OLMo, P. Walsh, L. Soldaini, and D. G. et. al. 2 olmo 2 furious, 2025.

[17] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms, 2017.

[18] M. Senghaas, F. Obeid, S. Jaghouar, W. Brown, J. M. Ong, A. Baker, J. Mattern, D. Auras, J. Straube, M. Basra, A. Ismail, and J. Hagemann. PRIME-RL: Async & decentralized RL training at scale. In *NeurIPS Workshop on GPU-Accelerated and Scalable Optimization*, 2025.

[19] A. Setlur, M. Y. R. Yang, C. V. Snell, J. Greer, I. Wu, V. Smith, M. Simchowitz, and A. Kumar. e3: Learning to explore enables extrapolation of test-time compute for LLMs. In *The Fourteenth International Conference on Learning Representations*, 2026.

[20] Z. Shao, P. Wang, Q. Zhu, R. Xu, J. Song, M. Zhang, Y. Li, Y. Wu, and D. Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024.

[21] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao. Reflexion: language agents with verbal reinforcement learning. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 8634–8652. Curran Associates, Inc., 2023.

[22] F. Tajwar, G. Zeng, Y. Zhou, Y. Song, D. Arora, Y. Jiang, J. Schneider, R. Salakhutdinov, H. Feng, and A. Zanette. Maximum likelihood reinforcement learning. In *The 1st Workshop on Scaling Post-training for LLMs*, 2026.

[23] L. von Werra, Y. Belkada, L. Tunstall, E. Beeching, T. Thrush, N. Lambert, S. Huang, K. Rasul, and Q. Gallouédec. TRL: Transformer Reinforcement Learning. https://github.com/huggingface/trl, 2020.

[24] X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh, H. H. Tran, F. Li, R. Ma, M. Zheng, B. Qian, Y. Shao, N. Muennighoff, Y. Zhang, B. Hui, J. Lin, R. Brennan, H. Peng, H. Ji, and G. Neubig. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*, 2025.

[25] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3–4):229–256, May 1992.

[26] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang. Demystifying llm-based software engineering agents. *Proc. ACM Softw. Eng.*, 2(FSE), June 2025.

[27] Y. E. Xu, Y. Savani, F. Fang, and Z. Kolter. Not all rollouts are useful: Down-sampling rollouts in llm reinforcement learning. *arXiv preprint arXiv:2504.13818*, 2025.

[28] Z. Xu, Y. Liu, Y. Yin, M. Zhou, and R. Poovendran. Kodcode: A diverse, challenging, and verifiable synthetic dataset for coding. *arXiv preprint arXiv:2503.02951*, 2025.

[29] A. Yang, A. Li, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Gao, C. Huang, C. Lv, C. Zheng, D. Liu, F. Zhou, F. Huang, F. Hu, H. Ge, H. Wei, H. Lin, J. Tang, J. Yang, J. Tu, J. Zhang, J. Yang, J. Yang, J. Zhou, J. Zhou, J. Lin, K. Dang, K. Bao, K. Yang, L. Yu, L. Deng, M. Li, M. Xue, M. Li, P. Zhang, P. Wang, Q. Zhu, R. Men, R. Gao, S. Liu, S. Luo, T. Li, T. Tang, W. Yin, X. Ren, X. Wang, X. Zhang, X. Ren, Y. Fan, Y. Su, Y. Zhang, Y. Zhang, Y. Wan, Y. Liu, Z. Wang, Z. Cui, Z. Zhang, Z. Zhou, and Z. Qiu. Qwen3 technical report, 2025.

[30] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. R. Narasimhan, and O. Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[31] Q. Yu, Z. Zhang, R. Zhu, Y. Yuan, X. Zuo, Y. Yue, W. Dai, T. Fan, G. Liu, L. Liu, X. Liu, H. Lin, Z. Lin, B. Ma, G. Sheng, Y. Tong, C. Zhang, M. Zhang, W. Zhang, H. Zhu, J. Zhu, J. Chen, J. Chen, C. Wang, H. Yu, Y. Song, X. Wei, H. Zhou, J. Liu, W.-Y. Ma, Y.-Q. Zhang, L. Yan, M. Qiao, Y. Wu, and M. Wang. Dapo: An open-source llm reinforcement learning system at scale, 2025.

[32] Y. Yuan, Y. Yue, R. Zhu, T. Fan, and L. Yan. What's behind ppo's collapse in long-cot? value optimization holds the secret, 2025.

[33] Y. Yue, Y. Yuan, Q. Yu, X. Zuo, R. Zhu, W. Xu, J. Chen, C. Wang, T. Fan, Z. Du, X. Wei, X. Yu, G. Liu, J. Liu, L. Liu, H. Lin, Z. Lin, B. Ma, C. Zhang, M. Zhang, W. Zhang, H. Zhu, R. Zhang, X. Liu, M. Wang, Y. Wu, and L. Yan. Vapo: Efficient and reliable reinforcement learning for advanced reasoning tasks, 2025.

[34] S. Zeng, Q. Wei, W. Brown, O. Frunza, Y. Nevmyvaka, Y. K. Zhao, and M. Hong. Reinforcing multi-turn reasoning in LLM agents via turn-level credit assignment. In *ICML 2025 Workshop on Computer Use Agents*, 2025.

[35] S. Zhang, Y. Dong, J. Zhang, J. Kautz, B. Catanzaro, A. Tao, Q. Wu, Z. Yu, and G. Liu. Nemotron-research-tool-n1: Tool-using language models with reinforced reasoning. *arXiv preprint arXiv:2505.00024*, 2025.

[36] C. Zheng, S. Liu, M. Li, X.-H. Chen, B. Yu, C. Gao, K. Dang, Y. Liu, R. Men, A. Yang, J. Zhou, and J. Lin. Group sequence policy optimization, 2025.

[37] L. Zhong, Z. Wang, and J. Shang. Debug like a human: A large language model debugger via verifying runtime execution step by step. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 851–870, 2024.

[38] A. Zhou, K. Yan, M. Shlapentokh-Rothman, H. Wang, and Y.-X. Wang. Language agent tree search unifies reasoning, acting, and planning in language models. In *Forty-first International Conference on Machine Learning*, 2024.

[39] R. Zhuang, T. Vu, A. Dimakis, and M. Sathiamoorthy. Improving multi-turn tool use with reinforcement learning, 2025. Accessed: 2025-04-17.

Appendix

1 Introduction **1**

2 Background: GRPO **3**

3 Proposed Method: MURPHY **3**

3.1 Pruning Strategies in MURPHY 5

4 Experiments **6**

4.1 Reward Propagation Ablation: MARS vs. MERS 8

4.2 Pruning Strategies Ablation: INTRAP vs. INTERP 8

5 Related Work **9**

6 Conclusion & Limitations **10**

A Extended Related Work **16**

A.1 Relation to Tree-Structured GRPO and Backup Theory 16

B GRPO: Objective and Additional Details **18**

C Multi-turn Rollout Formalism (Detailed) **18**

D Sensitivity to Multi-Iteration Scaffolds **21**

E Reflexion **21**

F Implementation Details **22**

F.1 Model Size and Compute Budget 22

F.2 Hyperparameters 22

F.3 Package Parameters 22

F.4 Additional Experiments 22

G Prompt Examples **26**

H Efficiency Gains **27**

I Broader Impact **28**

J Additional Ablations and Analyses **28**

J.1 Sensitivity to Feedback Format and Quality 28

J.2 Counterfactual Analysis of Retrospective Credit 29
J.3 Effect of Training-Time Rollout Depth 29
J.4 Failure-Gated Expansion and Training Cost 30
J.5 Scaffold-Controlled Baseline Comparisons 31

K Additional Generalization Experiments **32**
K.1 Application to Sudoku 32
K.2 Repository-Level Software Engineering 33

A Extended Related Work

LLM Agents for Software Development. Recent works [10, 37] have explored LLM agents for programming tasks such as code generation, bug fixing, and code migration. A key driver of progress in these domains has been inference-time iterative frameworks [21, 15], which leverage execution feedback to generate self-reflections for refining candidate programs [30, 26]. While these approaches underscore the value of iterative feedback and scaffolding, they primarily enhance the inference pipeline rather than the underlying model. Our work takes a complementary direction: we improve the reasoning and self-correction abilities of LLMs themselves through training-time optimization, thereby strengthening the base models that agentic frameworks depend on.

Reinforcement Learning with Verifiable Rewards for LLM Reasoning. Reinforcement learning (RL) is a popular paradigm for post-training LLMs to improve reasoning and align outputs with verifiable objectives. GRPO [20] revived interest in RL as an efficient alternative to PPO [17], offering comparable reasoning performance with far lower computational cost. Subsequent variants [33, 31, 32, 36] focus on stabilizing training, improving convergence, or shifting optimization from token-level to sequence-level. However, these algorithms remain tailored to single-turn tasks, optimizing models to produce one-shot completions without iterative refinement. Recent RLVR methods extend RL to multi-turn agents across domains such as search [4, 11], tool use [39, 35], and code generation [9, 8]. These methods typically compute advantage by summing outcome and turn-level rewards, which limits temporal credit propagation. In contrast, MURPHY delays credit assignment until a trajectory is complete, propagating rewards backward from successful states using a structured credit assignment criterion that preserves temporal consistency. Our work is most closely related to μCode [9] and RLEF [8], which also train LLMs with execution feedback. μCode jointly trains a generator and a learned verifier that scores multi-turn code solutions, while RLEF refines generations via PPO grounded in execution results. However, both approaches require auxiliary value functions or verifier LLMs, significantly increasing computational overhead and data acquisition cost. μCode further depends on its verifier at inference for Best-of-N selection, introducing additional latency. These design choices make direct comparison impractical and obscure the effect of the RL formulation itself. In contrast, MURPHY achieves comparable grounding in execution feedback and iterative refinement by extending GRPO to the multi-turn setting, while preserving its simplicity, efficiency, and architectural minimalism.

A.1 Relation to Tree-Structured GRPO and Backup Theory

TREEGRPO [7] is closely related in spirit to MURPHY: both methods replace trajectory-level credit with tree-structured credit assignment before applying a GRPO-style update. However, the two methods differ in the source of branching, the object being propagated, and the point at which normalization is applied. TREEGRPO targets diffusion/flow post-training, where branching occurs within a single denoising trajectory. It first computes normalized leaf advantages and then propagates them backward to obtain per-edge advantages using a log-probability weighted average over child branches. MURPHY instead targets multi-turn code generation, where branching is induced by executor feedback: failed programs create feedback-conditioned prompts whose descendants are later refinements. We therefore propagate execution rewards through the tree and then compute local group-relative advantages within each feedback-conditioned response group.

Tree-structured GRPO methods differ primarily in what quantity is backed up and how descendants are aggregated. TREEGRPO [7] propagates leaf advantages backward through a denoising tree using a policy-weighted average:

$$A(v) = \sum_{u \in C(v)} w_v(u) A(u), \qquad w_v(u) = \frac{\pi_{\theta_{\text{old}}}(u \mid v)}{\sum_{u' \in C(v)} \pi_{\theta_{\text{old}}}(u' \mid v)}.$$

In contrast, MURPHY propagates execution rewards through a feedback-conditioned refinement tree and computes local GRPO advantages only after this reward propagation step. This leads to two complementary backups:

$$\text{MERS}: \qquad \tilde{r}(v) = \frac{r(v) + \gamma \cdot \frac{1}{|C(v)|} \sum_{u \in C(v)} \tilde{r}(u)}{Z(v)},$$

and

$$\textsc{MARS}: \qquad \tilde{r}(v) = \max\left(r(v), \max_{u \in C(v)} \tilde{r}(u)\right).$$

Here $Z(v)$ denotes the local normalization factor used in the reward backup. Unlike the policy-weighted averaging rule in TREEGRPO, this normalization factor is tied to the realized refinement subtree below v. It may therefore depend on the number of sampled descendants, the rollout depth, and the feedback-induced branching pattern. Consequently, MERS should not be interpreted as the same estimator as the TREEGRPO advantage backup. Rather, it is a sample-dependent reward propagation rule that smooths execution rewards over the refinements actually generated under a given feedback context.

MERS is closest in spirit to the aggregation rule in TREEGRPO: both combine information from child branches before applying a GRPO-style update. The analogy, however, is only structural. In TREEGRPO, the backed-up quantity is a normalized advantage and the weights are induced by the old policy over denoising branches. In MURPHY, the backed-up quantity is an execution reward, and the normalization term $Z(v)$ is part of the sampled reward-propagation procedure.

Averaging over child branches can be viewed as a variance-reducing operation conditional on a fixed realized child set. Specifically, if $C(v)$ and weights w_v are fixed, and if child branch values X_u are conditionally independent with common variance σ^2, then

$$\mathrm{Var}\left[\sum_{u \in C(v)} w_v(u) X_u \,\middle|\, C(v), w_v\right] = \sigma^2 \sum_{u \in C(v)} w_v(u)^2.$$

For uniform weights this becomes $\sigma^2/|C(v)|$, conditional on the realized child set. Since the refinement tree, the number of descendants, and the normalization term $Z(v)$ are themselves determined by the sampled executor-feedback process, this conditional calculation should not be read as an unconditional variance or unbiasedness guarantee. Instead, it provides intuition for why MERS behaves as a smoothing backup over the realized refinement tree, whereas MARS behaves as an optimistic backup that propagates the best observed execution reward.

MARS, by contrast, is an optimistic best-descendant backup. For binary correctness rewards,

$$\mathbb{E}\left[\max_{u \in C(v)} R(u)\right] = \Pr(\exists u \in C(v) : R(u) = 1),$$

and, under conditional independence, this equals

$$1 - \prod_{u \in C(v)} (1 - p_u)$$

This objective is well aligned with sparse-reward code self-correction: a failed attempt should receive credit if the feedback it induces enables at least one later refinement to solve the task. Prior analysis of Monte Carlo tree-search backups similarly shows that max-style backups can preserve rare high-value trajectories that averaging may dilute [13].

This distinction also clarifies the roles of MERS and MARS. MERS is closest to TREEGRPO's aggregation rule: both estimate the value of an internal node by averaging over sampled descendants, and both can be interpreted as reducing variance relative to relying on a single sampled continuation. MARS, by contrast, is an optimistic best-descendant backup. This is motivated by the sparse and existential nature of code self-correction: a failed attempt can be useful if it exposes feedback that enables at least one later refinement to solve the task. Prior work on backup strategies in Monte Carlo tree search similarly observes that averaging can drown out rare high-value trajectories, while max-style backups can be beneficial when successful trajectories are scarce [13]. Thus, MERS provides a lower-variance expectation-like backup, whereas MARS is better aligned with best-of-k and pass@k style success criteria in sparse-reward code generation.

Recent work on MAXRL [22] provides a complementary perspective: in correctness-based tasks, likelihood-oriented objectives emphasize successful rollouts and higher-order pass@k events rather than only expected pass@1 reward. While MaxRL is developed for single-turn binary-outcome settings, it supports the intuition behind MARS: when success is rare, training signals that preserve and amplify successful descendants can be preferable to averaging them away. Our contribution is to instantiate this principle in a feedback-conditioned multi-turn GRPO setting, where earlier failed attempts are credited according to the downstream refinements they make possible.

B GRPO: Objective and Additional Details

Notation. We denote the model policy by $\pi_\theta(\cdot \mid \cdot)$ and the reference (older) policy by $\pi_{\theta_{\text{old}}}(\cdot \mid \cdot)$. Let G be the number of generations per prompt, $\mathcal{P}(Q)$ the distribution over input prompts/questions Q, and O the output space. For a given prompt $q \sim \mathcal{P}(Q)$, the reference policy produces a set of G responses, forming a response group $\{o_{q,1}, o_{q,2}, \ldots, o_{q,G}\}$. Each generation $o_{q,i} \in O$ corresponds to a full output trajectory, where $o_{q,i,t}$ denotes the t-th token and $o_{q,i,<t}$ the prefix up to (but excluding) token t. We write $|o_{q,i}|$ for the sequence length of the i-th generation. For a given prompt q, the reward model assigns a scalar score to each response in the group, yielding $\mathbf{r}_q = \{r_{q,1}, r_{q,2}, \ldots, r_{q,G}\}$. Moreover, for prompt q, the advantage associated with the t-th token of the i-th generation is defined as $\hat{A}_{q,i,t} = (r_{q,i} - \mu(\mathbf{r}_q))/\sigma(\mathbf{r}_q)$, where $\mu(\mathbf{r}_q)$ and $\sigma(\mathbf{r}_q)$ denote the mean and standard deviation of the group rewards, respectively. Finally, $D_{\text{KL}}(\pi_\theta || \pi_{\theta_{\text{ref}}})$ denotes the KL divergence between the current and reference policies, computed over all tokens in the generated sequences. The GRPO training objective is presented in Def. 1.

Definition 1. (GRPO Objective)

$$\mathcal{J}(\theta) = \mathbb{E}_{q \sim \mathcal{P}(Q),\ \{o_{(q,i)}\}_{i=1}^{G} \sim \pi_{\theta_{\text{old}}}(O|q)} \Bigg[\frac{1}{G} \sum_{i=1}^{G} \frac{1}{|o_{q,i}|} \sum_{t=1}^{|o_{q,i}|} \min \Big(R_\theta(q,i,t) \hat{A}_{q,i,t},$$
$$\text{clip}\Big(R_\theta(q,i,t),\ 1-\epsilon,\ 1+\epsilon \Big) \hat{A}_{q,i,t} \Big) \Bigg] - \beta D_{\text{KL}}(\pi_\theta \,\|\, \pi_{\text{ref}})$$

Where,

$$R_\theta(q,i,t) = \frac{\pi_\theta(o_{q,i,t} \mid q, o_{q,i,<t})}{\pi_{\theta_{\text{old}}}(o_{q,i,t} \mid q, o_{q,i,<t})}$$
$$\hat{A}_{q,i,t} = \frac{(r_{q,i} - \mu(\mathbf{r}_q))}{\sigma(\mathbf{r}_q)}$$

C Multi-turn Rollout Formalism (Detailed)

This appendix section provides a detailed treatment of the rollout-tree construction and indexing used throughout the main paper.

We define a feedback-conditioned rollout tree that captures how model generations evolve across multiple turns of interaction with the environment. Let s denote the turn index, S the total number of turns, and G_s the number of generations per prompt at turn s.

Turn 1: In the first turn ($s = 1$), the model receives an input prompt $q_{(1)}$ sampled from $\mathcal{P}(Q)$ and generates G_1 candidate programs which forms a response group $\{o_{\{q_{(1)},1\}}, o_{\{q_{(1)},2\}}, \ldots, o_{\{q_{(1)},G_1\}}\}$. Note that $\{o_{\{q_{(1)},j\}}\}_{j=1}^{G_1} \sim \pi_{\theta_{\text{old}}}(\cdot \mid q_{(1)})$. Each generation is executed against its associated test suite, producing a numerical reward $r_{\{q_{(1)},j\}}$ and environment feedback $f_{\{q_{(1)},j\}}$. The reward represents the proportion of test cases passed. The environment feedback contains the specific unit tests that passed or failed, along with any corresponding error messages. These G_1 generations form the first layer of output nodes in the rollout tree.

Turn 2: For each generation j in turn 1 that fails to achieve the maximum reward (which equals 1 in our setting, since it represents the proportion of test cases passed), the corresponding feedback is appended to the original prompt and the prior output from turn 1 to form a feedback-conditioned prompt: $q_{(2,j)} = [\, q_{(1)},\ o_{\{q_{(1)},j\}},\ f_{\{q_{(1)},j\}}\,]$ where $[\cdot]$ denotes textual concatenation. The model is then re-invoked to generate G_2 new candidate solutions: $\{o_{\{q_{(2,j)},k\}}\}_{k=1}^{G_2} \sim \pi_{\theta_{\text{old}}}(\cdot \mid q_{(2,j)})$. Each of these generations is evaluated to obtain $(r_{\{q_{(2,j)},k\}}, f_{\{q_{(2,j)},k\}})$, which denote the reward and feedback. These output generations represent refinements of their corresponding parent outputs $o_{\{q_{(1)},j\}}$ and collectively form the second layer of the rollout tree.

Turn s: Building on the previous turn, this procedure extends recursively to any turn $s \in \{1, \ldots, S-1\}$. We define $i_{[1:s]} = i_1, \ldots, i_s$ as a sequence of branch indices that trace a specific path through the tree, where i_j indicates the i'th candidate that was selected at turn j. Similarly, $i_{[1:1]}$ denotes

i_1. For each generation $o_{q_{(s,i_{[1:s-1]})},i_s}$ that fails to achieve the maximum reward, we construct a feedback-conditioned prompt:

$$q_{(s+1,i_{[1:s]})} = [q_{(s,i_{[1:s-1]})},\ o_{q_{(s,i_{[1:s-1]})},i_s},\ f_{q_{(s,i_{[1:s-1]})},i_s}]$$

where $[\cdot]$ denotes textual concatenation. The model is then re-invoked to generate G_{s+1} new candidate solutions:

$$\{o_{\{q_{(s+1,i_{[1:s]})},k\}}\}_{k=1}^{G_{s+1}} \sim \pi_{\theta_{\text{old}}}(\cdot \mid q_{(s+1,i_{[1:s]})})$$

Each candidate is evaluated to obtain its reward and feedback $(r_{\{q_{(s+1,i_{[1:s]})},k\}},\ f_{\{q_{(s+1,i_{[1:s]})},k\}})$. The resulting generations, $\{o_{\{q_{(s+1,i_{[1:s]})},k\}}\}_{k=1}^{G_{s+1}}$, form the child nodes of parent node $o_{\{q_{(s,i_{[1:s-1]})},i_s\}}$.

A complete path from the root (the initial prompt at turn 1) to a leaf at turn S (final output) can be expressed as:

$$q_{(1)} \to o_{q_{(1)},i_1} \to q_{(2,i_{[1:1]})} \to o_{q_{(2,i_{[1:1]})},i_2} \to \cdots \to o_{\{q_{(S,i_{[1:S-1]})},i_S\}}$$

where the indices $i_1, \ldots, i_S$ specify which branch is taken at each turn. Leaf nodes at turn S represent the final generations obtained after completing all refinement steps. Once rewards for all turns are computed, the rewards from these terminal nodes are propagated backward through their ancestors according to the credit assignment strategies described below.

Credit assignment formalism. After all outputs and corresponding rewards are generated and the rollout tree is constructed, we focus on assigning credit from later turns back to earlier ones. To achieve this, we explore two distinct strategies.

*Max Reward Strategy (*MARS*)*: The Max Reward Strategy is defined recursively, proceeding from the final turn back to the root. Since the final turn S has no children, the rewards at this turn remain unchanged. We then consider turn $s = S - 1$. Let $o_{\{q_{(s,i_{[1:s-1]})},i_s\}}$ denote a generation at turn $s = S - 1$ with an associated reward $r_{\{q_{(s,i_{[1:s-1]})},i_s\}}$. If this generation already achieves the maximum reward, it has no children; otherwise, its children are defined as:

$$C\big(o_{\{q_{(s,i_{[1:s-1]})},i_s\}}\big) = \{o_{\{q_{(s+1,i_{[1:s]})},1\}},\ \ldots,\ o_{\{q_{(s+1,i_{[1:s]})},G_S\}}\}$$

The corresponding set of rewards are defined as

$$C_r\big(o_{\{q_{(s,i_{[1:s-1]})},i_s\}}\big) = \{r_{\{q_{(s+1,i_{[1:s]})},1\}},\ \ldots,\ r_{\{q_{(s+1,i_{[1:s]})},G_S\}}\}$$

which defaults to zero if there are no children. We then update the reward as:

$$r_{\{q_{(s,i_{[1:s-1]})},i_s\}} = \max\left(r_{\{q_{(s,i_{[1:s-1]})},i_s\}},\ \max\Big(C_r(o_{\{q_{(s,i_{[1:s-1]})},i_s\}})\Big)\right)$$

This strategy assigns each node the maximum of its own reward and the best reward among its descendants. Intuitively, the descendant maximum represents the best outcome achievable through refinement, while taking the outer maximum ensures a node's credit never decreases; even when feedback fails to improve performance. This formulation captures the maximum progress achievable from any refinement path starting at that node. The procedure operates recursively in a backward pass: rewards are first updated for all nodes at turn $S - 1$ based on their children's values at turn S. This process continues backward through the tree, with each turn s receiving updated rewards based on the values from turn $s + 1$, until reaching the root.

*Mean Reward Strategy (*MERS*)*: The Mean Reward Strategy follows the same recursive credit assignment structure as the Max Reward Strategy (MARS), but differs in how rewards are propagated. Inspired by the return computation in REINFORCE [25], MERS updates each node's reward by incorporating the discounted *mean* of its children's rewards.

For a generation $o_{\{q_{(s,i_{[1:s-1]})},i_s\}}$ at turn s, let $\mathbb{I}_{\text{unsolved}}$ be an indicator that equals 1 if the problem remains unsolved at this node, and 0 otherwise. The update rule is:

$$r_{\{q_{(s,i_{[1:s-1]})},i_s\}} = \frac{r_{\{q_{(s,i_{[1:s-1]})},i_s\}} + \gamma \cdot \overline{C}_r\Big(o_{\{q_{(s,i_{[1:s-1]})},i_s\}}\Big)}{\mathbb{I}_{\text{unsolved}} \cdot (S - s) + 1}$$

where $\gamma \in [0, 1]$ is a discount factor controlling the influence of descendant rewards, and $\overline{C}_r(\cdot)$ denotes the mean reward over children of unsolved nodes (children of solved nodes do not exist since they represent terminal states).

The denominator equals $S - s + 1$ for unsolved nodes and 1 for solved nodes. This depth-based normalization serves two purposes: (1) it prevents rewards from growing unboundedly during backward propagation, as unsolved nodes can potentially accumulate discounted contributions from up to $S - s$ future turns; and (2) it ensures fair comparison between nodes that solve at different turns, a node that solves immediately at turn s retains its full reward, while a node that fails but has successful descendants receives appropriately scaled credit that accounts for the additional refinement steps required. All other aspects of the recursive procedure remain identical to MARS.

MARS *vs* MERS: MARS propagates the maximum descendant reward to the earlier turns, emphasizing peak performance, whereas MERS propagates the mean reward, emphasizing stability and overall consistency. MARS captures best-case improvement, while MERS provides a smoother estimate of expected progress. Together, they offer complementary views of feedback-driven credit assignment.

MURPHY *Objective*: Once the rewards are reassigned according to the chosen credit assignment strategy (MARS or MERS), advantages are computed similar to standard GRPO. The adjusted rewards serve as the basis for computing normalized advantages at each turn. Conditioned on a prompt $\tilde{q}$ and for G_s generations, we normalize each reward by subtracting the mean reward and dividing by the standard deviation of the rewards obtained across the G_s generations, yielding the normalized advantage $\hat{A}^{\text{MURPHY}}_{\tilde{q},i,t}$. Additionally, as defined earlier, each i-th generation at turn s, $o_{\{q_{(s,i_{[1:s-1]})},i\}} \in O$, corresponds to a complete output trajectory, where $o_{\{q_{(s,i_{[1:s-1]})},i,t\}}$ denotes the t-th token and $o_{\{q_{(s,i_{[1:s-1]})},i,<t\}}$ the prefix up to (but excluding) token t. We denote the sequence length of the i-th generation by $|o_{\{q_{(s,i_{[1:s-1]})},i\}}|$. At each turn, the GRPO objective is applied using these credit-adjusted advantages. This per-turn optimization allows MURPHY to incorporate feedback from later turns into earlier updates, effectively extending GRPO to a multi-turn setting. Finally, the divergence between the current and reference policies is captured by $D_{\text{KL}}(\pi_\theta \,\|\, \pi_{\text{ref}})$, computed over all tokens in the generated sequences. The resulting optimization objective, which integrates credit-assigned rewards, normalized advantages, and KL regularization at each turn, defines the MURPHY objective, distinguishing it from the standard GRPO formulation. The full MURPHY objective is presented in Def. 2.

Definition 2. (MURPHY Objective)

$$\mathcal{J}_{\text{MURPHY}}(\theta) = \mathbb{E}_{q\sim\mathcal{P}(Q)}\Big[\sum_{s,i_1,\ldots i_S} \mathcal{J}_{q_{(s,i_{[1:s]})}}(\theta)\Big]$$

Where the per prompt objective at each turn s is:

$$\mathcal{J}_{q_{(s,i_{[1:s-1]})}}(\theta) = \mathbb{E}_{\{o_{(\tilde{q},i)}\}_{i=1}^{G_s}\sim\pi_{\theta_{\text{old}}}(O|\tilde{q})}\left[\sum_{i=1}^{G_s}\frac{1}{G_s|o_{\{\tilde{q},i\}}|}\sum_{t=1}^{|o_{\{\tilde{q},i\}}|}\left(\min\left(R_\theta(\tilde{q},i,t)\hat{A}^{\text{MURPHY}}_{\tilde{q},i,t},\right.\right.\right.$$
$$\left.\left.\left.\text{clip}\left(R_\theta(\tilde{q},i,t),\,1-\epsilon,\,1+\epsilon\right)\cdot\hat{A}^{\text{MURPHY}}_{\tilde{q},i,t}\right) - \beta D_{\text{KL}}(\pi_\theta \,\|\, \pi_{\text{ref}})\right)\right]$$

$$\text{with, } \tilde{q} = q_{(s,i_{[1:s-1]})}$$

$$R_\theta(\tilde{q},i,t) = \frac{\pi_\theta(o_{\{\tilde{q},i,t\}} \mid \tilde{q}, o_{\{\tilde{q},i,<t\}})}{\pi_{\theta_{\text{old}}}(o_{\{\tilde{q},i,t\}} \mid \tilde{q}, o_{\{\tilde{q},i,<t\}})}$$

And, $\hat{A}^{\text{MURPHY}}_{\tilde{q},i,t}$ denotes the advantage.

Note. The design of MURPHY is broadly applicable across a range of RLVR algorithms, including PPO [17] and various extensions of GRPO [1, 31, 33]. In this work, we focus on GRPO due to its strong empirical performance in aligning LLMs [6, 29]. Extending MURPHY to other RLVR variants is conceptually straightforward, as it builds on the same underlying principles. While our experiments primarily focus on code generation, where rich, verifiable feedback is readily available, the framework can naturally extend to other domains such as mathematics or logical reasoning, provided suitable forms of feedback are accessible.

D Sensitivity to Multi-Iteration Scaffolds

MURPHY is explicitly designed to train models to incorporate environment feedback and self-correct over multiple turns. In multi-iteration evaluation—typical of agentic settings where feedback is present—MURPHY yields substantially larger gains. This behavior is expected and reflects the objective of MURPHY, which is to improve a model's ability to utilize feedback across turns rather than to optimize single-pass generation.

To assess sensitivity to the choice of multi-iteration scaffold, we conduct additional experiments using the Qwen3-1.7B model. We compare the base model, the GRPO-trained model, and the MURPHY-trained model under three distinct scaffolds: Reflexion [21], LATS [38], and DoT [15]. Importantly, no additional training is performed. We evaluate the same GRPO-MT checkpoint reported in Fig. 3 and the same MURPHY (MARS, INTRAP) checkpoint reported in Table 2. All scaffolds use their standard configurations with three iterations. We report mean accuracy and standard deviation over three independent runs on HumanEval.

Table 3: Sensitivity of Qwen3-1.7B performance on HumanEval to different multi-iteration scaffolds. No additional training is performed.

Model	Reflexion	LATS	DoT
Qwen3-1.7B (Base)	80.07 $\pm$ 1.27	78.95 $\pm$ 1.29	78.35 $\pm$ 3.88
Qwen3-1.7B GRPO-MT	81.14 $\pm$ 1.62	81.34 $\pm$ 2.51	82.92 $\pm$ 0.47
Qwen3-1.7B MURPHY (MARS, INTRAP)	**84.28 $\pm$ 2.01**	**86.89 $\pm$ 0.42**	**85.66 $\pm$ 0.43**

Across all scaffolds, the MURPHY-trained model consistently outperforms both the GRPO-MT trained model (trained under a matched compute budget) and the base model. These results indicate that the gains from MURPHY are not dependent on any particular inference-time scaffold, but instead reflect improved feedback utilization learned during training.

E Reflexion

Reflexion [21] is an inference-time iterative framework designed to improve reasoning through repeated interaction with feedback from an external environment. It employs three agents: an actor (M_a), an evaluator (M_e), and a self-reflection module (M_{sr}), which operate cyclically until a termination condition is met. For code generation, the process proceeds as follows:

1. **Actor step:** The actor M_a receives an input and generates an output (e.g., a code snippet).
2. **Evaluation step:** The evaluator M_e scores the output (e.g., the percentage of unit tests passed).
3. **Self-reflection step:** If the score is insufficient, the self-reflection module M_{sr} diagnoses the issue, proposes a fix, and appends both the failed output and the suggested correction to the input context. The updated input is then fed back to the actor, and the cycle repeats until either the task succeeds or a maximum number of iterations is reached.

In most implementations, the actor and the self-reflection module are instantiated by the same underlying language model. The self-reflection stage thus corresponds to the model reasoning over its own prior outputs augmented with feedback from the evaluator (or executor) and the previous input, output pairs, to generate improved responses in subsequent iterations.

F Implementation Details

We implement our framework on top of TRL [23], which provides efficient distributed training and a modular implementation of GRPO. We integrate TRL with vLLM for fast inference and large-scale rollout execution, enabling scalable multi-turn training in our experiments. Prompts used to train MURPHY are listed in App. G. All experiments use publicly available datasets. The base models (Qwen3, OLMo) are available for research use under the Apache 2.0 license.

F.1 Model Size and Compute Budget

All experiments were conducted on $8\times$ NVIDIA H100 GPUs. Our implementation builds on HuggingFace's TRL[3]. For efficiency, 2 GPUs were allocated for inference via vLLM, while the remaining 6 GPUs handled model updates. Checkpoints were saved every 50 steps, and for all baselines, we selected the checkpoint corresponding to one epoch.

F.2 Hyperparameters

We set the KL regularization coefficient to $\beta = 0.04$, the learning rate to 10^{-6}, and weight decay to 0.1 for both GRPO-MT and MURPHY variants. Unless otherwise specified, MURPHY uses two turns. In GRPO-MT, we sample 72 rollouts per prompt in turn 1; failed rollouts receive execution feedback and are extended in turn 2. For MURPHY, we sample 8 rollouts per prompt in turn 1 and up to 8 per prompt in turn 2, yielding at most 64 rollouts in the second turn. For μCODE and REVEAL we use the author recommended hyper-parameters for training.

Following [29], we use a temperature of 0.6 and top-p of 0.95 for all inference experiments.

F.3 Package Parameters

We use the Reflexion [21] framework to evaluate all trained models. The number of iterations is swept over $\{1, 3\}$, and `max-tokens` is set to each model's maximum generation length. Models are hosted via vLLM. Since all models fit on a single H100 GPU, we set `data-parallel-size` to 8 and enable prefix caching to accelerate evaluation. We use the following commands to install the appropriate packages:

```
pip install uv && \
uv pip install trl==0.19.1 && \
uv pip install gunicorn==20.1.0 && \
uv pip install fastapi==0.115.12
uv pip install uvicorn==0.34.2 && \
uv pip install aiohttp==3.11.18
uv pip install astunparse==1.6.3
uv pip install jsonlines tenacity && \
uv pip install vllm==0.8.5.post1
```

F.4 Additional Experiments

F.4.1 LiveCodeBench-v6 Per-Difficulty Results

Table 5 reports per-difficulty pass@1 on LiveCodeBench-v6 using Qwen3-4B as the base model. Base, GRPO-MT, μCODE, and MURPHY are evaluated under the multi-turn Reflexion framework at 1, 3, and 5 iterations, while REVEAL is evaluated using its official generation-verification scaffold with the same inference-turn budgets. This gives REVEAL its intended evaluation setting: although its generation prompt, output format, and parser differ from Reflexion, the scaffold is structurally similar in that it conditions on prior incorrect responses and environment feedback over multiple turns to refine subsequent attempts. MURPHY demonstrates superior performance in multi-turn (Iter-3/5) settings. On the Hard subset, gains over the Base model increase with iteration (+3.57, +5.10, +5.14 pp at Iter-1/3/5). The only exception is the Medium subset at Iter-1, where μCODE outperforms MURPHY by 1.67 pp; however, MURPHY surpasses all baselines by Iter-3 (+3.85 pp)

[3] https://huggingface.co/docs/trl/en/index

Table 4: Performance of OLMo-2-1124-7B-Instruct, Qwen3-1.7B and Qwen3-4B variants on HumanEval and MBPP benchmarks. GRPO-MT corresponds to the multi-turn baseline. MURPHY (Ours) is highlighted. Δ_3 denotes the difference between the **Iter-3** performance of the GRPO-MT/μCODE/MURPHY-trained models and that of **Base (Iter-3)**, within each model block; green indicates improvement (darker = larger gain), red indicates regression.

Model	HumanEval (%)			MBPP (%)		
	Iter-1	Iter-3	Δ_3	Iter-1	Iter-3	Δ_3
OLMo-2-1124-7B-Instruct						
Base	37.20 ± 0.86	46.04 ± 0.43	–	19.90 ± 0.42	29.60 ± 0.28	–
GRPO-MT	39.02 ± 0.00	41.87 ± 0.93	–4.17	28.33 ± 0.31	33.60 ± 0.69	+4.00
μCode [9]	35.17 ± 0.35	41.46 ± 1.06	–4.58	22.47 ± 0.42	31.60 ± 0.60	+2.00
MURPHY - MARS	**45.53 ± 0.70**	**52.24 ± 1.96**	+6.20	29.33 ± 0.90	**39.67 ± 1.29**	+10.07
Qwen3-1.7B						
Base	74.19 ± 1.95	80.07 ± 1.27	–	43.47 ± 0.64	53.93 ± 3.52	–
GRPO-MT	70.33 ± 0.93	82.11 ± 0.35	+2.04	**46.00 ± 0.40**	58.53 ± 1.03	+4.60
μCode [9]	73.58 ± 1.27	83.34 ± 0.70	+3.27	44.00 ± 0.69	56.20 ± 0.60	+2.27
MURPHY - MARS	**79.67 ± 3.01**	**86.58 ± 1.06**	+6.51	44.73 ± 0.50	**62.00 ± 1.91**	+8.07
Qwen3-4B						
Base	90.04 ± 3.13	93.49 ± 0.93	–	52.13 ± 0.42	70.93 ± 1.01	–
GRPO-MT	89.00 ± 1.50	93.80 ± 1.00	+0.31	51.50 ± 0.40	71.40 ± 0.70	+0.47
μCode [9]	87.81 ± 2.65	94.10 ± 1.54	+0.61	50.87 ± 0.46	71.93 ± 0.12	+1.00
MURPHY - MARS	**92.48 ± 0.61**	**95.73 ± 0.31**	+2.24	**53.33 ± 1.15**	**73.33 ± 1.31**	+2.40

and further extends the margin at Iter-5 (+4.38 pp). This trend supports our claim that MURPHY's training objective compounds across self-correction iterations.

Table 5: **Per-difficulty pass@1 on LiveCodeBench-v6 with Qwen3-4B.** Values are mean$_{\pm\text{std}}$ over 3 runs, in percent. **Bold** marks the best method per row; underline marks the second best. MURPHY achieves competitive or superior performance in multi-turn setting.

Subset	Iter	Base	GRPO-MT	μCODE	REVEAL	MURPHY (Ours)
Easy	1	$84.47_{\pm 2.04}$	$84.47_{\pm 0.93}$	$85.09_{\pm 0.62}$	$86.23_{\pm 1.09}$	$\mathbf{87.49_{\pm 0.52}}$
	3	$92.34_{\pm 1.26}$	$92.13_{\pm 0.79}$	$92.55_{\pm 0.31}$	$92.49_{\pm 0.56}$	$\mathbf{93.38_{\pm 0.18}}$
	5	$93.27_{\pm 0.89}$	$91.92_{\pm 0.83}$	$93.62_{\pm 0.18}$	$93.79_{\pm 0.54}$	$\mathbf{94.23_{\pm 0.16}}$
Medium	1	$45.29_{\pm 3.46}$	$47.91_{\pm 0.53}$	$\mathbf{52.79_{\pm 0.66}}$	$46.51_{\pm 1.53}$	$51.12_{\pm 0.53}$
	3	$58.38_{\pm 1.84}$	$59.51_{\pm 1.06}$	$61.23_{\pm 1.28}$	$62.38_{\pm 2.58}$	$\mathbf{66.23_{\pm 0.54}}$
	5	$61.69_{\pm 1.44}$	$60.56_{\pm 1.86}$	$63.19_{\pm 0.84}$	$61.61_{\pm 1.74}$	$\mathbf{67.57_{\pm 0.61}}$
Hard	1	$10.76_{\pm 0.59}$	$11.05_{\pm 1.00}$	$11.95_{\pm 2.64}$	$11.43_{\pm 1.59}$	$\mathbf{14.33_{\pm 0.41}}$
	3	$16.19_{\pm 1.72}$	$18.29_{\pm 0.58}$	$18.10_{\pm 1.08}$	$18.13_{\pm 1.28}$	$\mathbf{21.29_{\pm 0.43}}$
	5	$19.15_{\pm 1.03}$	$19.14_{\pm 0.99}$	$20.09_{\pm 1.46}$	$19.14_{\pm 0.86}$	$\mathbf{24.29_{\pm 0.38}}$

To test whether the LiveCodeBench-v6 gains observed on Qwen3-4B also hold for a smaller model, we evaluate Qwen3-1.7B under the same iterative execution-feedback protocol. As shown in Tab. 6, Iter-1 serves primarily as a sanity check: MURPHY does not degrade one-shot generation and achieves the best aggregate performance among all methods. The main comparison is Iter-3, where models can condition on execution feedback and refine earlier attempts. In this multi-turn setting, MURPHY improves over the strongest non-MURPHY baseline by +1.93 pp on the aggregate split and by +3.40 pp on the Medium split. On the Hard split, MURPHY matches μCode while exhibiting lower variance. These results show that MURPHY's LiveCodeBench-v6 gains are not specific to

Qwen3-4B and that its advantage is most apparent in the feedback-conditioned multi-turn setting targeted by our method.

Table 6: **LiveCodeBench-v6 results on Qwen3-1.7B.** We report pass@1 on the Easy, Medium, Hard, and aggregate splits under 1-turn and 3-turn evaluation. Iter-1 serves as a sanity check for one-shot generation, while Iter-3 is the primary multi-turn setting for evaluating feedback-conditioned self-correction. MURPHY achieves the best aggregate performance at both settings, with the largest gains appearing at Iter-3. Results are averaged over three runs; standard deviations are shown after $\pm$.

Iter.	Variant	Easy	Medium	Hard	Aggregate (All)
1	BASE	71.95 ± 1.09	25.57 ± 0.66	3.90 ± 0.33	32.54 ± 0.19
	GRPO-MT	71.74 ± 0.54	30.36 ± 0.46	4.19 ± 0.33	34.32 ± 0.24
	μCode	73.50 ± 2.00	33.07 ± 0.16	4.95 ± 0.44	36.08 ± 0.67
	MURPHY	$\mathbf{74.43 \pm 1.60}$	$\mathbf{34.90 \pm 0.66}$	$\mathbf{6.09 \pm 0.83}$	$\mathbf{37.41 \pm 0.52}$
3	BASE	80.12 ± 0.94	33.51 ± 1.71	5.72 ± 0.50	38.52 ± 1.06
	GRPO-MT	80.64 ± 1.77	39.35 ± 1.23	6.48 ± 0.59	41.05 ± 0.05
	μCode	82.61 ± 3.89	42.06 ± 0.40	$\mathbf{8.86 \pm 0.50}$	43.42 ± 1.14
	MURPHY	$\mathbf{84.88 \pm 1.00}$	$\mathbf{45.46 \pm 1.34}$	$\mathbf{8.86 \pm 0.29}$	$\mathbf{45.35 \pm 0.81}$

F.4.2 Ablation: Effect of Training Dataset Size

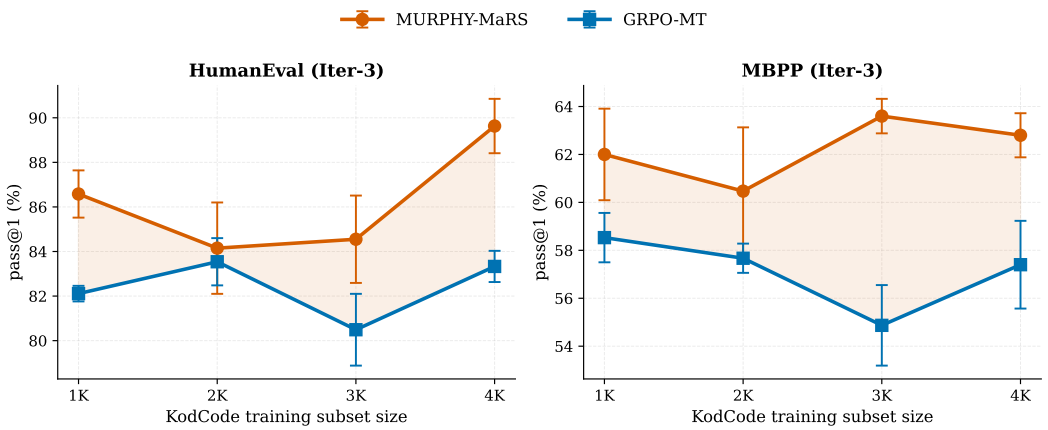


Figure 4: **MURPHY's multi-turn advantage holds across training data scales.** Qwen3-1.7B trained on nested KodCode subsets (1K $\subset$ 2K $\subset$ 3K $\subset$ 4K) and evaluated under three-iteration Reflexion (Iter-3) on HumanEval (left) and MBPP (right). Markers show the mean over 3 seeds; error bars indicate one standard deviation. Shaded regions highlight where MURPHY-MARS exceeds GRPO-MT. MURPHY outperforms compute-matched GRPO-MT at every dataset scale on both benchmarks. We focus on the multi-turn (Iter-3) setting here since it directly probes the credit-assignment mechanism that distinguishes MURPHY from GRPO-MT.

To examine whether MURPHY's gains hold beyond the 1K training split used in our main experiments (Fig. 3), we construct three additional nested subsets of KodCode (2K $\subset$ 3K $\subset$ 4K) and train Qwen3-1.7B with both GRPO-MT and MURPHY-MARS under matched compute budgets. Fig. 4 reports Iter-3 (multi-turn) pass@1 on HumanEval and MBPP across all four scales, including the 1K result from Fig. 3 for completeness. Although neither method scales monotonically with dataset size, MURPHY-MARS outperforms GRPO-MT at every scale on both benchmarks, with average gains of $+3.9$ pp on HumanEval and $+5.1$ pp on MBPP. This pattern indicates that MURPHY's multi-turn advantage is a property of the credit-assignment mechanism rather than an artifact of the specific training subset used in our main results.

F.4.3 Ablation: Effect of Pruning in Multi-Turn MURPHY Training

We noted in the main paper that increasing the number of turns in MURPHY can lead to exponential growth in computational cost. To mitigate this, we design and evaluate two pruning strategies. In this experiment, we extend our setup to a 3-turn setting and study the effect of pruning on performance. Results in Tab. 7 show that the pruned variant achieves competitive or even superior performance compared to the non-pruned counterpart.

Table 7: Ablation study comparing pruning versus non-pruning strategies for Qwen3-1.7B trained with MURPHY for 3 turns on the KodCode dataset. Reported numbers indicate *pass@1* (%) over three independent runs. Pruned variant achieves competitive performance compared to non-pruned MURPHY. Although the MBPP Iter-3 means round to the same values as the two-turn pruning ablation in Tab. 2, these results come from separate three-turn training runs; the different standard deviations reflect different seed-level outcomes.

Model	**HumanEval**		**MBPP**	
	Iter-1	**Iter-3**	**Iter-1**	**Iter-3**
MURPHY- MARS, INTERP	78.66 ± 2.66	$\mathbf{84.15 \pm 1.83}$	$\mathbf{45.53 \pm 1.17}$	61.07 ± 0.42
MURPHY- MARS	$\mathbf{79.27 \pm 2.11}$	83.94 ± 1.41	44.80 ± 1.11	$\mathbf{62.20 \pm 1.40}$

G Prompt Examples

To train the MURPHY objective, we employ the following prompts. The system prompt is used at each dialogue turn, and the feedback prompts are applied during every feedback turn.

System Prompt

You are a Python coding AI agent. When given a Python function signature and docstring, you must provide a complete Python solution following this exact format:

1. **Reasoning Phase:** Use `<think>...</think>` tags to contain your complete thought process.
   - Break down the problem requirements step by step
   - Identify key constraints, edge cases, and potential pitfalls
   - Plan your algorithm and data structures
   - Walk through examples to validate your approach
   - Explain your logic thoroughly as if working on scratch paper
2. **Implementation Phase:** Use `<output>...</output>` tags to contain your final code.
   - Include the complete function with the original signature
   - Ensure your code directly implements the approach from your thinking
   - Write clean, readable code with appropriate comments if needed

Your solution must be complete, correct, and handle all specified requirements.

Feedback Header

You have previously attempted this problem `{num-attempts}` time(s).

Feedback Body

Previous Attempt #`{num-attempts}` **Analysis:**

Your earlier thought process and implementation:

Let me think step by step.
`<think>`
`{code-generated}`

Test Case Results:
- Passed: `{passed-tests}`/`{total-tests}` test cases
- Failed: `{failed-tests}`/`{total-tests}` test cases

Detailed Feedback: `{feedback-string}`

Note: For failed test cases, your code's output is shown followed by a '#' symbol indicating the failure.

Feedback Footer.

Instructions for Your Next Attempt:

1. **Failure Analysis Phase:**
   (a) Carefully examine each failed test case to understand exactly what went wrong
   (b) Identify the specific lines of code or logic that caused the failures

(c) Look for patterns across multiple failed cases (e.g., all involve negative numbers, empty inputs, etc.)
(d) Determine if failures stem from algorithmic errors, edge case handling, or implementation bugs

2. **Solution Refinement Phase:**
   (a) Build upon any correct aspects of your previous solution that passed tests
   (b) Redesign the problematic parts of your algorithm to handle the failed cases
   (c) Ensure your new approach covers edge cases that weren't properly handled before
   (d) Consider additional edge cases that might not be in the test suite but could break your solution
3. **Implementation Phase:**
   (a) Write your improved solution that directly addresses the identified failure points
   (b) Test your logic mentally against the failed cases to verify it would now pass
   (c) Ensure your solution maintains correctness for previously passing test cases

Use the concrete feedback from your `{num-attempts}` previous attempt(s) to create a more robust and accurate solution.

H Efficiency Gains

Practical cost of rollout expansion. Although the number of nodes in an unpruned feedback-conditioned tree can grow with the turn budget, several factors reduce the realized cost in practice. Successful candidates terminate immediately, and only failed candidates are expanded. In addition, descendants of the same parent share substantial prompt prefixes, including the original problem, previous output, and executor feedback. In our experiments, we use vLLM for rollout generation [14]. Beyond this setup, the shared-prefix structure could be further exploited by additional systems techniques such as prefix caching, asynchronous rollout generation [18], and speculative or parallel decoding [3]. These optimizations are not used in our experiments and are complementary to MURPHY's pruning, which targets post-rollout optimization cost rather than rollout construction.

Table 8: **Where pruning saves compute.** Average per-step timing breakdown across an end-to-end training run for unpruned MURPHY vs. MURPHY with INTERP pruning on Qwen3-1.7B (8×H100s, up to 72 rollouts per prompt; 8 in turn-1 and up to 64 in turn-2). Pruning is applied post-rollout, so generation and reward times are essentially unchanged; the speedup concentrates in the optimization phase, where the number of sequences entering gradient computation is reduced.

Phase	**Unpruned**	**INTERP**	**Δ**
Generation (vLLM)	30.54s	30.46s	−0.3%
Reward (code exec)	4.86s	4.47s	−8.0%
Rollout overhead†	19.86s	17.75s	−10.6%
Rollout total	55.25s (76%)	52.67s (92%)	−4.7%
Optimization	17.49s (24%)	4.53s (8%)	**−74.1%**
Total	**72.74s**	**57.20s**	**−21.4%**

† Reference log-probs, tokenization, and advantage computation.

Measured pruning speedup. To localize the speedup from pruning, we profile per-step timing averaged across an end-to-end INTERP run against unpruned MURPHYunder matched configurations (Qwen3-1.7B, 8×H100 GPUs, 72 rollouts per prompt: 8 in turn 1 and up to 64 in turn 2); see Tab. 8. Generation with vLLM and reward computation via code execution are essentially unchanged (−0.3% and −8.0%, respectively), since pruning is applied *after* rollouts are collected. The savings instead concentrate in optimization, where pruning reduces the number of sequences entering gradient computation: optimization time drops by **74.1**%, yielding a **21.4**% reduction in average per-step

Table 9: Inference-time feedback ablation with Qwen3-1.7B trained using MURPHY-MaRS. Pass@1 is in percent and Δ is the reported improvement from Iter-1 in percentage points. The full-feedback row is the paper reference.

	HumanEval		MBPP	
Feedback	Iter-3	Δ	Iter-3	Δ
Binary pass/fail	84.55 ± 1.27	$+4.88 \pm 0.61$	50.87 ± 0.83	$+4.07 \pm 0.31$
Aggregate score	84.15 ± 2.11	$+4.47 \pm 0.70$	51.13 ± 0.61	$+4.33 \pm 0.12$
Natural-language error category	84.15 ± 0.61	$+4.47 \pm 2.31$	50.87 ± 1.55	$+4.07 \pm 0.99$
Corrupted executor output	84.76 ± 1.83	$+5.08 \pm 0.35$	55.00 ± 1.44	$+8.20 \pm 0.87$
No executor feedback	85.16 ± 1.86	$+5.49 \pm 0.00$	50.33 ± 0.83	$+3.53 \pm 0.31$
Same-task shuffled feedback	86.18 ± 0.93	$+6.50 \pm 0.93$	58.60 ± 1.22	$+11.80 \pm 0.72$
Full executor output (paper reference)	86.58 ± 1.06	$+6.91$	62.00 ± 1.91	$+17.27$

training time. This confirms that INTERP's computational benefit comes from avoiding gradient updates on uninformative trajectories, not from shortening rollouts. This is important because aggressive rollout-time pruning could discard feedback signals that later turns rely on.

I Broader Impact

While MURPHY introduces some new dynamics through iterative self-correction and reflective optimization, the associated risks appear modest overall. The main considerations involve ensuring that feedback loops remain interpretable and that reward signals do not inadvertently reinforce narrow or heuristic reasoning. There is also some potential for subtle reward hacking, where the model optimizes for easily verifiable but shallow improvements, or for mild distributional drift if reflective heuristics fail to generalize beyond training contexts. Nonetheless, because MURPHY still relies on verifiable rewards and bounded reflection, these risks are relatively contained and can be mitigated through careful evaluation design, human oversight, and robust validation across diverse task settings.

J Additional Ablations and Analyses

J.1 Sensitivity to Feedback Format and Quality

We evaluate how the information supplied during inference-time refinement affects a Qwen3-1.7B model trained with MURPHY-MaRS. The evaluation covers all 164 HumanEval tasks and 500 MBPP tasks, with final programs scored on hidden tests. We compare full executor output with six alternatives: binary pass/fail feedback; the number and percentage of visible tests passed; a coarse natural-language error category; corrupted executor output; no executor feedback; and full feedback from another failed candidate for the same task. The coarse categories are timeout, syntax error, runtime error, and wrong result. Full feedback includes the visible-test pass count and raw executor output, such as failed assertions, exceptions, and observed values. Feedback perturbations affect the text presented to the model. The execution and hidden-test evaluation procedures are unchanged.

Table 9 reports Iter-3 Pass@1 and the reported improvement over Iter-1. The six feedback interventions are shown alongside the full-feedback reference from the paper. Results are means and sample standard deviations over three independent seeds; uncertainty for the full-feedback improvement was not supplied.

All six interventions improve over their respective initial solutions, but the no-feedback condition also improves substantially. Additional generation and self-refinement therefore account for part of the observed gain. Coarse feedback does not consistently outperform this control: on HumanEval, binary, aggregate, and categorical feedback yield lower mean scores than no feedback. On MBPP, corrupted and same-task shuffled feedback reach 55.00% and 58.60%, respectively, compared with 50.33% without executor feedback. This suggests that diagnostic information can remain useful when incomplete or misaligned with the current candidate. The full-feedback reference has the highest reported mean score on both benchmarks. These results concern the use of perturbed feedback

Table 10: Independent holdout comparisons for 186 MaRS-credited failures. Counts pool all three seeds. Rates reproduce the reported summaries across seeds and therefore need not equal the pooled proportions. A dash indicates that a seed-level summary was not supplied.

Comparison outcome	Count	Rate (%)
Best-of-eight visible-test success		
Feedback succeeds; no feedback fails	97	52.3 ± 6.1
Tie	86	46.1 ± 4.7
Both conditions succeed	48	—
Both conditions fail	38	—
No feedback succeeds; feedback fails	3	1.6 ± 1.6
Fraction of eight revisions passing all visible tests		
Higher with feedback	127	68.4 ± 5.2
Equal across conditions	51	—
Lower with feedback	8	4.3 ± 0.8

at inference time; they do not establish robustness to incorrect tests or to training with corrupted rewards.

J.2 Counterfactual Analysis of Retrospective Credit

MaRS can credit a failed attempt when at least one of its descendants succeeds. To examine whether executor feedback contributes to these successes, we evaluate the trained Qwen3-1.7B MaRS model on 50 held-out MBPP problems using seeds 123, 124, and 125. For each problem and seed, we sample eight initial solutions and generate eight feedback-conditioned revisions for each failed initial solution. MaRS assigns retrospective credit when at least one revision passes all visible tests. This occurs for 64 of 212, 62 of 194, and 60 of 208 failed initial solutions across the three seeds, respectively, yielding 186 credited cases among 614 failures.

For each credited case, we generate two fresh holdout sets of eight revisions: one conditioned on the problem, failed solution, and executor feedback, and one conditioned on the same problem and failed solution without executor feedback. We do not reuse the descendants that triggered credit assignment, since doing so would guarantee a feedback-conditioned success by construction. Both conditions are evaluated on visible tests.

Feedback yields a uniquely successful best-of-eight outcome in 97 cases, whereas the reverse occurs in only three cases (Table 10). It also increases the fraction of passing revisions in 127 of 186 cases and decreases it in eight. Thus, the feedback advantage frequently reappears in independent samples rather than being confined to the descendants used to select credited failures. These measurements are conditional on MaRS-credited cases and hold the failed parent fixed. They quantify the contribution of feedback in this selected set, rather than the overall rate of incorrect credit assignments or the causal necessity of the parent attempt.

Qualitative repair taxonomy. We additionally inspect 20 comparisons, deliberately selecting ten in which feedback increases the passing fraction and ten in which it does not. Table 11 summarizes six descriptive categories. Labels were generated by GPT-5.6-sol from the failed solution, executor feedback, revisions from both conditions, and their test outcomes. The analysis is illustrative: the balanced selection does not estimate category prevalence, and no human inter-annotator agreement is claimed.

J.3 Effect of Training-Time Rollout Depth

We compare MURPHY-MaRS trained with maximum rollout depths of two and three turns. Table 12 reports Iter-1 and Iter-3 evaluation scores as means and standard deviations over three independent evaluation runs. HumanEval and MBPP use Qwen3-1.7B; the LiveCodeBench-v6 results are aggregated over the Easy, Medium, and Hard subsets.

Table 11: Illustrative taxonomy of 20 deliberately balanced feedback/no-feedback comparisons. Counts describe this selected sample only.

Category	Count	Representative behavior
Feedback increases the passing fraction		
Output-contract repair	5	Corrects printing versus returning, tuple structure, or element ordering.
Algorithmic or semantic repair	4	Uses a failed assertion to correct the task interpretation or recurrence.
Runtime or API localization	1	Repairs a missing or incorrectly named entry point.
Feedback does not increase the passing fraction		
Repair is not feedback-specific	2	Both conditions independently discover the same correction.
Feedback is ignored or partially used	6	Preserves the error, changes only a comment, or makes an incomplete repair.
Algorithmic misconception persists	2	Reacts to the failed example without correcting the underlying algorithm.

Table 12: Two-turn versus three-turn MURPHY-MaRS training. Scores are Pass@1 (%); differences are three-turn minus two-turn means in percentage points. Numerical entries are reproduced from the depth comparison.

Benchmark	Training depth	Iter-1	Iter-3
HumanEval	Two turns	79.67 ± 3.01	86.58 ± 1.06
	Three turns	79.27 ± 2.11	83.94 ± 1.41
	Difference	-0.40	-2.64
MBPP	Two turns	44.74 ± 1.91	62.00 ± 1.91
	Three turns	44.80 ± 1.11	62.20 ± 1.40
	Difference	$+0.06$	$+0.20$
LiveCodeBench-v6	Two turns	37.41 ± 0.52	45.35 ± 0.81
	Three turns	40.61 ± 0.89	46.97 ± 0.28
	Difference	$+3.20$	$+1.62$

Increasing training depth does not yield uniform improvements. MBPP changes little, while HumanEval decreases by 2.64 percentage points at Iter-3. In contrast, the reported LiveCodeBench-v6 aggregate increases by 3.20 points at Iter-1 and 1.62 points at Iter-3. The results indicate that additional training turns can help in some settings, but do not establish monotonic benefits from deeper trees. The marginal value of depth depends on the model, task distribution, and opportunities for further correction. Since equal total training compute is not established for this comparison, it should not be interpreted as a compute-controlled estimate of the effect of depth alone.

J.4 Failure-Gated Expansion and Training Cost

Generation savings from failure-gated expansion. MURPHY terminates nodes that attain maximum reward and expands only failed nodes. This reduces generation and execution before optimization, whereas INTERP selects among already collected rollouts and reduces optimization cost. Across 200 logged steps of a two-turn MaRS run on KodCode-1K, the fraction of first-turn nodes attaining maximum reward increases from 64.58% during steps 1–50 to 73.71% during steps 151–200. Correspondingly, the fraction expanded to turn two decreases from 35.42% to 26.29%. Over the full run, 69.96% terminate immediately and 30.04% are expanded.

With eight initial candidates and eight children per failed candidate, the mean number of generated nodes per prompt is

$$\overline{N}_{\text{gated}} = 8 + 8 \times 8 \times 0.3004 = 27.2256, \tag{8}$$

compared with $8+8\times8 = 72$ for unconditional expansion. Thus, failure-gated expansion reduces the number of generated and executed nodes by approximately 62.19%. This is a node-count reduction,

Table 13: Qwen3-1.7B training profile on eight B200 GPUs, averaged over 50 steps. Times are seconds per step except for the last row. The reported per-rollout normalization uses 216 rollouts per step.

Component	Unpruned MURPHY-MaRS	GRPO-MT
Rollout generation	43.7	58.0
First-turn generation	25.9	28.9
Second-turn generation	17.8	29.0
Code execution	4.2	3.7
Optimization and overhead	21.5	17.4
Total per step	69.4	79.1
B200 GPU-seconds per rollout	2.57	2.93

not a measured wall-clock speedup. These logs characterize expansion from turn one to turn two and do not measure the expansion rate at greater depths.

Controlled component-wise profiling. We profile unpruned MURPHY-MaRS and GRPO-MT using Qwen3-1.7B on eight NVIDIA B200 GPUs, averaging measurements over 50 training steps. GRPO-MT generates 36 first-turn responses per prompt and one second-turn refinement per response, totaling 72 nodes per prompt. MURPHY generates eight first-turn responses and up to eight children per failed response, giving a maximum of 72 nodes per prompt. The profile reports a budget of 216 rollouts per step with a prompt batch size of three. INTERP is disabled and all generated MURPHY nodes are retained for optimization. GRPO-MT trains on linear trajectories without MURPHY's branching refinement groups or retrospective reward propagation.

In this profile, unpruned MURPHY requires 69.4 seconds per step, compared with 79.1 seconds for GRPO-MT, a reduction of approximately 12.3%. The difference comes primarily from rollout generation; code execution and optimization are slightly more expensive for MURPHY. Groups of children share a feedback-conditioned parent context, which permits prefix-cache reuse and is a plausible contributor to the generation-time difference. The contribution of caching is not separately ablated. These measurements exclude INTERP and should not be directly compared with the main paper's H100 timings. At Iter-3, the corresponding Qwen3-1.7B scores are 86.58% versus 82.11% on HumanEval and 62.00% versus 58.53% on MBPP for MURPHY and GRPO-MT, respectively (Table 15).

J.5 Scaffold-Controlled Baseline Comparisons

REVEAL under native and Reflexion evaluation. To examine the effect of the inference scaffold, we evaluate REVEAL with Qwen3-4B using both its native generation–verification harness and the Reflexion harness used for MURPHY. Table 14 reports LiveCodeBench-v6 results by difficulty and inference iteration budget. MURPHY has a higher mean score than both REVEAL configurations in every reported setting. At Iter-5, its margins over the stronger REVEAL configuration are 0.44, 5.96, and 3.99 percentage points on Easy, Medium, and Hard tasks, respectively. These results support the comparison under a common inference scaffold without implying identical training cost across methods.

Common-harness comparison across benchmarks. We additionally compare the base model, GRPO-MT, μCode, REVEAL, and MURPHY using Qwen3-1.7B and the Reflexion harness on HumanEval, MBPP, and the aggregate LiveCodeBench-v6 split. All trained methods use the same KodCode-1K examples; REVEAL and μCode use their method-specific training configurations. Iter-3 is the primary evaluation point for feedback-conditioned refinement, and Iter-1 is included for completeness. Table 15 reports means and sample standard deviations over three runs.

At Iter-3, MURPHY exceeds the strongest comparator by 3.24 percentage points on HumanEval, 3.47 points on MBPP, and 1.93 points on LiveCodeBench-v6. Its Iter-1 advantage is not universal: on MBPP, GRPO-MT attains 46.00%, compared with MURPHY's 44.73%. The results therefore support improved multi-turn refinement under a common evaluation harness, without implying uniformly stronger initial generations.

Table 14: Qwen3-4B on LiveCodeBench-v6 under alternative REVEAL harnesses. Values reproduce the reported mean and uncertainty in percent. Δ is MURPHY's mean minus the larger REVEAL mean in each row (pp).

Difficulty	Iter.	REVEAL native	REVEAL Reflexion	MURPHY	Δ
Easy	1	86.23 ± 1.09	84.1 ± 1.1	87.49 ± 0.52	$+1.26$
	3	92.49 ± 0.56	92.9 ± 1.1	93.38 ± 0.18	$+0.48$
	5	93.79 ± 0.54	92.8 ± 1.1	94.23 ± 0.16	$+0.44$
Medium	1	46.51 ± 1.53	48.2 ± 0.5	51.12 ± 0.53	$+2.92$
	3	62.38 ± 2.58	60.7 ± 0.9	66.23 ± 0.54	$+3.85$
	5	61.61 ± 1.74	61.3 ± 1.3	67.57 ± 0.61	$+5.96$
Hard	1	11.43 ± 1.59	10.8 ± 1.5	14.33 ± 0.41	$+2.90$
	3	18.13 ± 1.28	18.5 ± 0.8	21.29 ± 0.43	$+2.79$
	5	19.14 ± 0.86	20.3 ± 1.6	24.29 ± 0.38	$+3.99$

Table 15: Common Reflexion evaluation with Qwen3-1.7B. Values are Pass@1 (%), reported as mean $\pm$ sample standard deviation over three runs. Δ is the Iter-3 improvement over the base model on the same benchmark (pp).

Benchmark	Method	Iter-1	Iter-3	Δ
HumanEval	Base	74.19 ± 1.95	80.07 ± 1.27	—
	GRPO-MT	70.33 ± 0.93	82.11 ± 0.35	$+2.04$
	μCode	73.58 ± 1.27	83.34 ± 0.70	$+3.27$
	REVEAL	72.76 ± 3.36	82.72 ± 3.07	$+2.65$
	MURPHY	79.67 ± 3.01	86.58 ± 1.06	$+6.51$
MBPP	Base	43.47 ± 0.64	53.93 ± 3.52	—
	GRPO-MT	46.00 ± 0.40	58.53 ± 1.03	$+4.60$
	μCode	44.00 ± 0.69	56.20 ± 0.60	$+2.27$
	REVEAL	44.40 ± 0.72	56.67 ± 1.72	$+2.74$
	MURPHY	44.73 ± 0.50	62.00 ± 1.91	$+8.07$
LiveCodeBench-v6	Base	32.54 ± 0.19	38.52 ± 1.06	—
	GRPO-MT	34.32 ± 0.24	41.05 ± 0.05	$+2.53$
	μCode	36.08 ± 0.67	43.42 ± 1.14	$+4.90$
	REVEAL	34.44 ± 1.12	41.27 ± 0.76	$+2.75$
	MURPHY	37.41 ± 0.52	45.35 ± 0.81	$+6.83$

K Additional Generalization Experiments

K.1 Application to Sudoku

We apply MURPHY to 9×9 Sudoku using Qwen3-1.7B. The trainer, two-stage rollout procedure, MaRS credit assignment, optimization settings, and prompt structure are retained from the code experiments. The domain interface changes: verification checks rows, columns, boxes, and fixed clues; the reward is the fraction of blank cells correctly filled; and feedback lists violated constraints and affected cells. Feedback is computed solely from the puzzle and submitted grid and does not expose reference-solution information. Models are trained on 2,048 puzzles and evaluated on a disjoint set of 320 puzzles using the same iterative evaluation protocol.

MURPHY exceeds GRPO-MT by 20.63 percentage points at Iter-1 and 34.58 points at Iter-2 (Table 16). The comparison uses the same reported two-stage rollout budget, while GRPO-MT assigns trajectory-level rewards without retrospective propagation. MURPHY also benefits from additional inference iterations beyond its two-stage training horizon, reaching 97.60% at Iter-3 and 98.44% at Iter-5, compared with 64.27% and 68.23% for GRPO-MT. This experiment demonstrates applying the training method in a non-code domain with immediate, programmatically verifiable feedback.

Table 16: Qwen3-1.7B on 320 held-out Sudoku puzzles. Entries are the evaluation percentages reported in the experiment. The rollout column reproduces the reported training rollouts per step.

Method	Rollouts	Iter-1	Iter-2	Iter-3	Iter-4	Iter-5
Base (no RL)	—	2.08	3.44	4.17	4.79	5.62
Single-turn GRPO	8	36.88	40.42	42.81	44.90	47.29
GRPO-MT	72	43.33	58.13	64.27	67.08	68.23
MURPHY	72	63.96	92.71	97.60	98.12	98.44

Table 17: Preliminary SWE-bench Verified evaluation with Qwen3-4B and mini-swe-agent, using a 32K context window and at most 50 agent iterations.

Method	Resolved tasks	Resolved (%)
Base	14/500	2.80
GRPO-MT	12/500	2.40
MURPHY	26/500	5.20

It involves Sudoku-specific training and does not constitute zero-shot transfer of a code-trained checkpoint.

K.2 Repository-Level Software Engineering

We evaluate the Qwen3-4B checkpoint used in the main code-generation experiments on all 500 SWE-bench Verified tasks using the mini-swe-agent harness, a 32K-token context window, and a maximum of 50 agent iterations per task. This setting requires repository inspection, execution, and iterative modification beyond isolated function generation.

MURPHY resolves 26 tasks, compared with 14 for the base model and 12 for GRPO-MT (Table 17), corresponding to improvements of 2.40 and 2.80 percentage points. These preliminary results suggest that the learned refinement behavior can transfer to longer repository-level interactions. They evaluate longer inference-time interaction and do not establish the training-time scalability of substantially deeper rollout trees.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The main claims in the abstract and introduction accurately reflect the paper's contributions and scope, as they are supported by the proposed method in Sec. 3, the experimental results in Sec. 4, and the conclusion and limitations discussion in Sec. 6.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: The paper discusses limitations of the work in Sec. 6, specifically in the conclusion and limitations discussion.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [N/A]

Justification: [N/A]

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: The paper provides information needed to reproduce the main experimental results in the experimental setup in Sec. 4 and the additional implementation and inference details in the appendix, along with the prompts used for the experiments.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [No]

Justification: While all data and models used in the paper are open-source, and the training recipe is described in Sec. 4 and the appendix, the code will be released upon acceptance.

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (https://neurips.cc/public/guides/CodeSubmissionPolicy) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (https://neurips.cc/public/guides/CodeSubmissionPolicy) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

Answer: [Yes]

Justification: The paper specifies the experimental settings in Sec. 4, including benchmarks, models, evaluation protocol, and repeated runs, with further details provided in the appendix.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [Yes]

Justification: The paper reports mean and standard deviation across runs in the experimental results in Sec. 4, including the main figures and tables.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.

- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: The paper discusses compute resources and training-time costs in Sec. 4 and provides additional efficiency and implementation details in the appendix.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The paper's experiments use standard code-generation benchmarks, public models, and public datasets, as described in Sec. 4 and the appendix, and do not involve human-subject data collection or sensitive data.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

Justification: The paper discusses broader risks and practical implications in the broader impact or discussion material in the appendix, including concerns around robustness, misuse, and deployment.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

Answer: [N/A]

Justification: The paper does not release a high-risk pretrained model, image generator, scraped dataset, or other asset requiring special safeguards; the work is evaluated using existing models and benchmarks as described in Sec. 4 and the appendix.

Guidelines:

- The answer [N/A] means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [N/A]

Justification: The paper does not introduce or release a new dataset, model, or code asset; the experiments rely on existing benchmarks, datasets, and model families described in Sec. 4 and the appendix.

Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [N/A]

    Justification: The paper does not introduce or release a new dataset, model, or code asset; the experiments rely on existing benchmarks, datasets, and model families described in Sec. 4 and the appendix.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing or human-subject experiments; the evaluation is based on code-generation benchmarks described in Sec. 4.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [N/A]

Justification: The paper does not involve human-subject research, as the experiments use code-generation benchmarks described in Sec. 4, so IRB approval is not applicable.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [Yes]

Justification: The core method directly involves LLMs, since the paper trains and evaluates code-generation language models using feedback-conditioned multi-turn rollouts and GRPO-based optimization. Claude Code was used as a coding assistant during implementation and debugging, but all methodological decisions, experimental design, validation, and reported results were determined and verified by the authors.

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.