MURPHY: Reflective Multi-Turn Reinforcement Learning for Self-Correcting Code Generation in Large Language Models

Chanakya Ekbote[*]
cekbote@mit.edu
AWS AI LABS

Vijay Lingam[†]
vjlingam@amazon.com
AWS AI LABS

Behrooz Omidvar-Tehrani
AWS AI LABS

Jun (Luke) Huan
AWS AI LABS

Sujay Sanghavi
AMAZON

Stefano Soatto
AWS AI LABS

Abstract

Reinforcement Learning with Verifiable Rewards (RLVR) has emerged as a powerful framework for enhancing the reasoning capabilities of large language models (LLMs). However, existing approaches such as Group Relative Policy Optimization (GRPO) and its variants, while effective on reasoning benchmarks, struggle with agentic tasks that require iterative decision-making. We introduce MURPHY, a multi-turn reflective optimization framework that extends GRPO by incorporating iterative self-correction during training. By leveraging both quantitative and qualitative execution feedback, MURPHY enables models to progressively refine their reasoning across multiple turns. Evaluations on code generation benchmarks with model families such as Qwen and OLMo show that MURPHY consistently improves performance, achieving up to a 8% relative gain in pass@1 over GRPO, on similar compute budgets.

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*

A growing body of work explores large language models (LLMs) as software engineering agents that interact with their environment through code execution and feedback [18, 12, 13]. Rather than producing a single static response, these systems operate within agentic scaffolds [26, 33] that guide iterative reasoning and allow LLMs to act, observe, and improve over multiple rounds of interaction. For example, in a typical coding agnetic scaffold [18, 12], the agent generates a solution by performing a single or series of actions, and executes it for evaluation, often through unit tests or other automated checks. When execution fails, the agent receives feedback such as error messages, stack traces, or failing test cases, and is re-prompted with the original task, its previous attempt, and the new

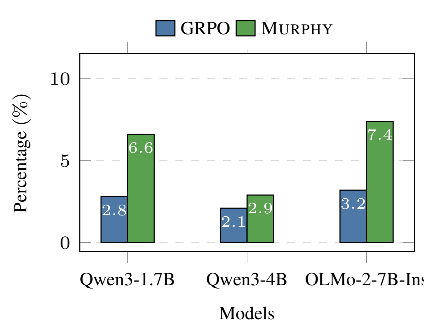

Figure 1: MURPHY-trained models solve up to 4.2% more problems than GRPO. See Tab. 1.

[*]Work performed while the author was a research intern at AWS AI Labs.
[†]Corresponding author

39th Conference on Neural Information Processing Systems: ForLM Workshop (NeurIPS 2025).

feedback. This process continues for multiple turns until the model succeeds or reaches a fixed iteration limit. These systems highlight the growing ability of LLMs to reason and adapt through environmental feedback at inference time. In code generation tasks [9], such feedback naturally arises from execution logs, compiler errors, or test results. However, these methods [18, 12] remain fundamentally *training-free*: they improve model behavior through structured inference via feedback rather than through parameter updates.

On the other hand, training methodologies such as Reinforcement Learning with Verifiable Rewards (RLVR) have enabled a new generation of language models [15, 6, 19, 24] to exhibit strong reasoning capabilities across mathematics, coding, and general problem-solving. Recent RLVR algorithms, including Group Relative Policy Optimization (GRPO) [17] and its extensions [29, 27, 22], have become dominant approaches for post-training LLMs on verifiable reasoning tasks. However, GRPO and its extensions are fundamentally designed for a single-turn setting: they optimize model behavior using an isolated prompt–response–reward tuple, with no notion of multi-turn interaction defined in their objective. Here, a turn denotes one cycle in which the model receives a prompt (which includes feedback from its previous response), reasons based on the prompt, and generates a revised output. Taken together, these limitations underscore a methodological gap: *while inference-time agentic frameworks exploit iterative feedback to refine reasoning, GRPO optimizes solely from terminal rewards on a given prompt, without leveraging intermediate environmental feedback*. This motivates the following key research question.

> **Key Research Question:** How can RLVR algorithms be extended to integrate iterative feedback, and to what extent does this improve reasoning and performance within agentic frameworks?

To address this question, we propose MURPHY, a novel RLVR algorithm that extends GRPO to a multi-turn setting by conditioning optimization on intermediate environmental feedback. Extending GRPO beyond a single turn is non-trivial: it requires defining how rewards obtained in later turns should be propagated backward to earlier attempts, so that intermediate reasoning and output turns that initially failed but ultimately led to success through feedback receive appropriate credit. At the first turn, MURPHY generates G_1 generations per prompt and computes group-based rewards as in GRPO. In subsequent turns, generations that fail to achieve the maximum reward, e.g., those failing unit tests or producing incorrect outputs, are refined using signals from the environment such as executor logs or test results. This feedback is appended to the original prompt, and the model is re-prompted to generate a new batch of G_s generations (where s denotes the turn) conditioned on the combined context (prompt, previous output, and feedback), repeating this process for a fixed number of turns. Rewards from successful final-turn rollouts are then propagated backward to earlier turns using MURPHY's credit-assignment criterion, allowing partially correct but improving attempts to receive credit. To manage the computational cost of multi-turn updates, MURPHY employs pruning strategies that retain only promising trajectories while bounding total gradient updates per turn. In summary, our main contributions are:

> **Main Contributions.**
>
> 1. We introduce MURPHY, a multi-turn RLVR algorithm that extends GRPO to leverage execution feedback for grounded reasoning and self-correction in code generation tasks. (Sec. 4)
> 2. We design pruning mechanisms that enable scalable multi-turn RLVR training (MURPHY) while preserving performance. (Subsec. 4.1)
> 3. We evaluate MURPHY on three code generation benchmarks across two model families (OLMo, Qwen) and sizes (1.7B–7B), achieving up to $+5\%$ pass@1 over GRPO baselines. (Sec. 5)

2 Related Work

LLM Agents for Software Development. Recent studies [9, 32] investigate LLM agents for code generation, bug fixing, and code migration. A central factor behind their progress is inference-time iterative frameworks [18, 12], which leverage execution feedback and self-reflection to refine candidate programs [25, 21]. 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.

RVLR for LLM Reasoning. RL has emerged as a powerful paradigm for aligning LLMs with verifiable objectives. GRPO [17] renewed interest in RL as an efficient alternative to PPO [16], achieving comparable reasoning performance with lower computational cost. Follow-up variants [29, 27, 28, 31] improve stability, convergence, or shift optimization from token-level to sequence-level, yet they remain tailored to single-turn tasks. Our work is closely related to μCode [8] and RLEF [7], which incorporate execution feedback during training. μCode jointly trains a generator with a learned verifier that scores multi-turn code solutions, whereas RLEF applies PPO grounded in execution results. However, both rely on auxiliary value functions or verifier LLMs, increasing computational and data costs. In contrast, MURPHY extends GRPO to the multi-turn setting, achieving comparable grounding in execution feedback while retaining simplicity, efficiency, and architectural minimalism. See App. A for extended related work.

3 Background: GRPO

Group Relative Policy Optimization (GRPO; [17]) is a variant of Proximal Policy Optimization (PPO) designed to improve the efficiency and stability of policy updates in LLM fine-tuning. Unlike PPO, which estimates advantages using a learned value function (critic), GRPO replaces the critic with an empirical baseline: the mean reward across all generations produced by the model for the same prompt. In both methods, rewards are provided by a reward model. Specifically, for each input prompt, the model samples a set of G candidate responses, forming a *response group*. The reward model assigns a score to each response, and advantages are computed by standardizing rewards within the group: subtracting the group mean and dividing by the group standard deviation. This yields relative, normalized advantage values. As in PPO, GRPO may incorporate a penalty term to prevent the updated policy from drifting too far from the reference policy, typically enforced via a Kullback–Leibler (KL) divergence regularizer to ensure stable updates. A formal definition of the GRPO objective, along with additional details, is provided in App. B.

4 Proposed Method: MURPHY

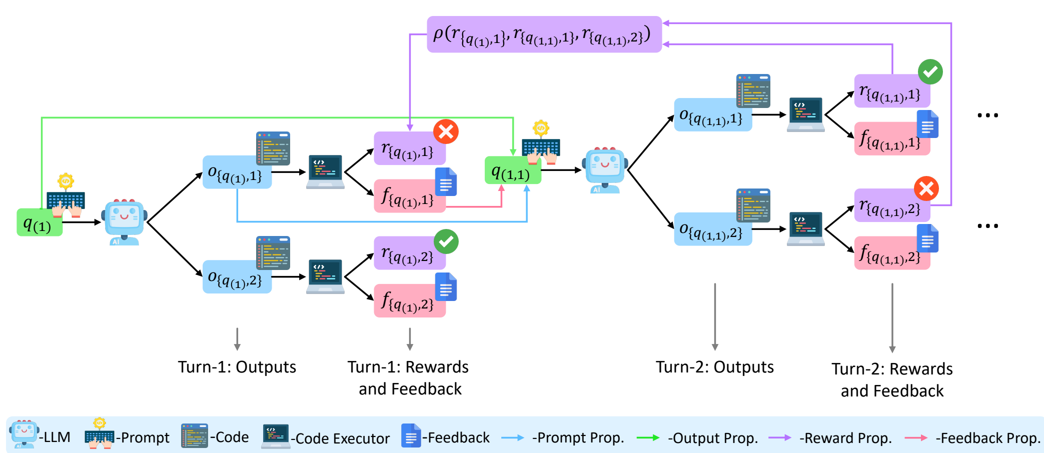

Figure 2: Overview of MURPHY. Given an input prompt (q), G code generations (o) are generated and evaluated using a reward function (r). Generations that do not achieve the maximum reward are revised based on executor feedback (f), combining the original prompt with the failed output, and re-prompted to generate another G candidates. This iterative process continues for a fixed number of turns, with rewards from later turns propagated backward. The example illustrates the case with $G = 2$, where G represents the number of rollouts per prompt, and $\rho(\cdot)$ denotes the credit assignment strategy.

Extending GRPO to multi-turn interaction requires formalizing how environmental feedback informs optimization. We study a feedback-rich code generation setting where model outputs are executed and scored against test suites, yielding two feedback types: (1) quantitative (e.g., proportion of test cases passed) and (2) qualitative (e.g., error traces or failing cases). Murphy extends GRPO by introducing (i) a multi-turn rollout mechanism that conditions generation on feedback, and (ii) a credit-assignment scheme that propagates rewards from later successful turns to earlier attempts.

Multi-turn rollout: For each task prompt, the policy begins at turn 1 by generating G_1 candidate solutions, each evaluated for reward against the prompt's test suite. Failed generations are paired with their corresponding feedback and, together with the original prompt and prior outputs, form new conditioning contexts for subsequent turns. At turn s, the model generates G_s new candidates for each failed case, continuing this feedback–refinement process for a fixed number of turns. This per-prompt, iterative rollout enables progressive improvement conditioned on feedback. Unlike standard GRPO, MURPHY *delays advantage computation until the full rollout is complete*, allowing final rewards to retroactively shape credit assignment.

Credit assignment: Once rewards for all turns are computed, MURPHY propagates future rewards backward to earlier turns using a temporal credit-assignment scheme. Although early generations may achieve low initial rewards, incorporating execution feedback in later turns often leads to successful refinements. To ensure that these intermediate steps are properly credited, rewards from later successful generations are distributed to earlier ones according to their contribution to eventual success. After reward redistribution, advantages are computed for each generation, and the MURPHY loss is applied to update the policy. Together, these mechanisms extend GRPO to handle multi-turn, feedback-conditioned optimization. Below, we formalize the *multi-turn rollout mechanism* and *credit-assignment* process and define the MURPHY objective, extending GRPO to handle multi-turn feedback.

Notation and formalism. 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 $\mathcal{P}(Q)$ the distribution over input prompts/questions Q, and O the output space. As described earlier, in the first turn, the model generates G_1 candidate solutions for each prompt. For generations that fail, feedback is obtained from the environment. The model is then re-prompted with the original prompt, its previous output, and the corresponding feedback to produce G_s new generations per prompt at turn s. This iterative procedure naturally forms a tree structure, where the root corresponds to the original prompt, and each subsequent generation (augmented with feedback) forms a child node, with generations at turn $s+1$ linked to their parent at turn s (see Fig. 2). We formalize this multi-turn rollout process as follows.

Multi-turn rollout formalism. 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: At 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 $\{o_{\{q_{(1)},j\}}\}_{j=1}^{G_1} \sim \pi_{\theta_{\text{old}}}(\cdot \mid q_{(1)})$. For any generation indexed by j, the generated code is executed against its associated test suite, producing a numerical reward $r_{\{q_{(1)},j\}}$ and environment feedback $f_{\{q_{(1)},j\}}$. In our setting, the reward represents the proportion of test cases passed, while the environment feedback consists of the specific unit tests that passed or failed, along with any corresponding error messages. These generations together 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 $s \in \{1, \ldots, S-1\}$. Note that we define:

$$i_{[1:s]} = i_1,\ \ldots,\ i_s$$

For each generation $o_{\{q_{(s,i_{1:(s-1)})},i_s\}}$ that fails to achieve the maximum reward, the feedback-conditioned prompt is constructed:

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

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 generation is evaluated to obtain $(r_{\{q_{(s+1,i_{[1:s]})},k\}},\ f_{\{q_{(s+1,i_{[1:s]})},k\}})$, which denote the corresponding reward and feedback, respectively. The resulting generations, $\{o_{\{q_{(s+1,i_{[1:s]})},k\}}\,\}_{k=1}^{G_{s+1}}$ form the children nodes of the unique corresponding parent node $o_{\{q_{(s,i_{1:(s-1)})},i_s\}}$. Hence, a complete path from the root (at the prompt of turn 1) to a leaf at stage S (the output of the model), is expressed as:

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

Note that 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 parent nodes according to the credit assignment strategies described below.

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

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

The corresponding set of rewards are defined as

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

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 rule selects the greater value between a node's own reward and the maximum reward among its descendants. Intuitively, the inner maximum represents the best improvement achievable through feedback, while the outer maximum guarantees that a node's reward does not decrease when feedback fails to yield better performance. This formulation therefore captures the maximum attainable progress achievable from feedback at each turn. The procedure is applied recursively: rewards are first updated for all generations at turn $S-1$, then propagated backward toward the root. At each turn s, rewards are reassigned based on the updated values from turn $s+1$, continuing until all turns have been processed.

*Mean Reward Strategy (*MERS*)*: The Mean Reward Strategy follows the same recursive credit assignment structure as the Max Reward Strategy (MaRS), with the key difference lying in how rewards are propagated. Instead of using the maximum reward among descendants, MeRS updates each node's reward using the *mean* of its child rewards, normalized over the number of stages the update has taken place over. Specifically, for a generation $o_{\{q_{(s,i_{1:(s-1)})},i_s\}}$ at stage s, the update rule is,

$$r_{\{q_{(s,i_{1:(s-1)})},i_s\}} = \frac{1}{S-s+1}\left(r_{\{q_{(s,i_{1:(s-1)})},i_s\}} + \gamma \cdot \text{mean}\Big(C_r(o_{\{q_{(s,i_{1:(s-1)})}\}})\Big)\right)$$

where γ is a discount factor controlling the influence of descendant rewards, and the normalization term $\frac{1}{S-s+1}$ ensures that credit assignment remains consistent across varying depths of the tree. Intuitively, this formulation captures the average improvement achieved through feedback, scaled by how many stages of updates have occurred. The normalization prevents early-stage nodes (with many descendants) from accumulating disproportionately large credit, while ensuring that terminal nodes, having no children, are not penalized. All other aspects of the recursive procedure, including the bottom-up updates from stage $S-1$ to the root, remain identical to those in MaRS.

MARS *vs* MERS: MaRS propagates the maximum descendant reward to the earlier stages, 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 stage 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. 1.

Definition 1. (MURPHY Objective)

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

Where the per prompt objective at each turn s is:

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

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 [16] and various extensions of GRPO [1, 27, 29]. In this work, we focus on GRPO due to its strong empirical performance in aligning LLMs [6, 24]. 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.

4.1 Pruning Strategies in MURPHY

By default, MURPHY sets $G_s = G$ for all turns, maintaining a fixed number of generations per prompt throughout the multi-turn process. However, this multi-turn setup introduces significant computational cost. In the worst case, when success is achieved only at the final turn S, the number of generations per prompt can grow exponentially to G^S, resulting in a large search tree and substantial memory overhead. This makes MURPHYcomputationally expensive to optimize. While system-level optimizations such as vLLM with paged attention and KV caching [11] make the generation process relatively efficient, the optimization phase remains costly. Large-scale rollouts can quickly exhaust GPU memory, and batching schemes that treat each generated sample as an independent training batch tend to be prohibitively slow. To address these challenges, we introduce two *pruning strategies* that reduce the number of rollouts at each turn, thereby making MURPHY computationally tractable without compromising performance. We describe these strategies in detail below.

*Intra-Group Pruning (*INTRAP*)*: In INTRAP, pruning operates recursively: starting from the children of stage $S-1$, each pruning step is followed by reward propagation as described in Sec. 4, and the process continues backward until reaching the root. This ordering—pruning before reward reassignment—ensures that only informative trajectories are retained for credit propagation across turns. At each turn s, given a pruning budget b, we retain only the b trajectories conditioned on the prompt whose rewards contribute most to the total reward variance within that group. The remaining trajectories, along with all their descendants, are discarded, and the process proceeds recursively until the root is reached. This approach is inspired by Xu et al. [22], who demonstrate that retaining trajectories with the highest reward variance within a group can reduce optimization cost while maintaining performance comparable to GRPO. We extend this principle to the multi-turn setting of MURPHY.

*Inter-Group Pruning (*INTERP*)*: In INTERP, the goal is to prune entire groups of children corresponding to generations conditioned on a given prompt. Specifically, each generation at a turn has a group of children, and we decide how many of these groups to retain based on a pruning budget b. To rank the groups, we assign a score inspired by the UCB sampling strategy [2], defined as $\alpha_1\mu + \alpha_2\sigma$, where μ and σ denote the mean and standard deviation of the rewards within each group of children. The groups are ranked in descending order of this score, and only the top b groups are retained, with b determining the available computational budget. The remaining groups and their descendants are discarded. Similar to IntraP, this process is applied recursively and proceeds backward from the latter turns to the earlier ones. At each level, pruning is performed first, followed by credit assignment. In our experiments, we set $\alpha_1 = 0$ and $\alpha_2 = 1$, which effectively prioritizes groups exhibiting higher reward variance. Intuitively, a higher standard deviation not only captures greater variability within a group but also indicates that the model is uncertain about that group's performance. Such groups likely contain some generations with near-maximum rewards, making them valuable targets for further optimization. In contrast, groups where all rewards are uniformly high (near 1) or uniformly low (near 0) provide limited learning signal, as the model either has already mastered or completely failed the underlying behavior. High-variance groups, therefore, represent the most informative regions for continued improvement.

5 Experiments

In this section, we first provide an overview of the models and datasets used in our experiments, followed by a detailed description of the setup and results, including ablation studies. We provide implementation details in App. D.

Models. We evaluate our framework using two open-source model families: Qwen3 (1.7B, 4B) [24] and OLMo2 (OLMo-2-1124-7B-Instruct) [14]. These models differ in both architecture and scale, enabling us to assess the generality of our method across diverse settings.

Datasets, Metrics, and Evaluation. We detail the datasets, metrics, and evaluation procedures employed in our experiments below.

Training Dataset: We fine-tune all models on 1,000 samples randomly drawn from the KodCode dataset [23]. This dataset was chosen due to its minimal overlap with evaluation benchmarks[3]. We further study the impact of training data size in Appendix D.4.2.

Evaluation Datasets. We evaluate trained models on a suite of programming and reasoning benchmarks, including HumanEval [4], MBPP [3], and BigCodeBench-Hard [35]. For BigCodeBench-Hard, which does not include visible unit tests, we randomly sample two test cases from the full test suite to construct visible tests.

Metrics and Evaluation Protocol. Reflexion [18] is a widely adopted inference-time iterative framework that enhances reasoning by refining incorrect outputs through feedback and self-generated reflections (see App. C for details). To assess reasoning refinement and self-correction, we integrate all models into the Reflexion framework and report *pass@1* under two settings: (i) *Single iteration*, equivalent to standard input–output prompting, and (ii) *Three iterations*, where feedback from visible test cases is incorporated into subsequent prompts. The agent terminates once all visible tests pass or when the maximum iteration limit 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.

5.1 MURPHY EXPERIMENTS

We evaluate three models, Qwen3 (1.7B, 4B) [24] and OLMo-2-1124-7B-Instruct [14], each fine-tuned on 1,000 samples randomly drawn from the KodCode dataset [23]. Evaluation is conducted on the benchmark datasets described in Sec. 5 using the Reflexion framework. We report *pass@1* performance under both single- and multi-iteration settings, as summarized in Tab. 1. Additional results comparing with a naive multi-turn extension are provided in Appendix D.4.1, and experiments with higher multi-turn training are included in Appendix D.4.3. For all our tables, best results are **bold** and second best are <u>underlined</u>. Results are reported as pass@1 accuracy (% mean $\pm$ stdev).

Reflexion: Single-iteration setting. (Iter-1 in Tab. 1) Models trained with the GRPO objective consistently outperform their base counterparts, achieving notable gains; for instance, a $\sim 3\%$ improvement on HumanEval for OLMo-2-1124-7B-Instruct. MURPHY achieves competitive or superior performance than GRPO in this setting.

Reflexion: Multi-iteration setting. (Iter-3 in Tab. 1) We repeat the experiments with three iterations in the Reflexion framework to assess self-correction and reasoning-refinement capabilities. Increasing the number of iterations leads to consistent performance improvements across all models and benchmarks. Models trained with MURPHY surpass both GRPO-trained and base models, achieving gains of up to 8% over GRPO. These results demonstrate the effectiveness of multi-turn reflective optimization in enhancing reasoning refinement and self-correction.

5.2 Ablation 1: MARS vs. MERS

As described in Sec. 4, we compare two reward propagation strategies: MARS and **MERS**. We train Qwen3-1.7B and OLMo-2-1124-7B-Instruct on 1,000 KodCode samples under each strategy and report results in Tab. 2. Across both models and multiple Reflexion iterations, MARS consistently matches or surpasses MERS, independent of the discount factor γ. The key difference lies in handling non-binary rewards: MERS averages rewards across generations, diluting the learning signal when few outputs score highly, whereas MARS propagates the strongest outcome, allowing rare but valuable high-reward trajectories to dominate the update. This makes MARS particularly effective in multi-turn settings with sparse rewards, while the gap between the two strategies diminishes in binary-reward tasks.

5.3 Ablation 2: INTRAP vs. INTERP

In Subsec. 4.1, we introduced two pruning strategies: Intra-Group (INTRAP) and Inter-Group (INTERP) pruning. Both variants prune the rollouts tree, thereby reducing the total number of gradient

[3]Refer to Section 3.2 of KodCode [23] for details on contamination analysis.

Model	Rollouts	HumanEval (%)		MBPP (%)		BigCodeBench (%)	
		Iter-1	Iter-3	Iter-1	Iter-3	Iter-1	Iter-3
Qwen3-1.7B							
Base	–	74.19 ± 1.95	80.07 ± 1.27	43.47 ± 0.64	53.93 ± 3.52	7.20 ± 2.73	18.24 ± 1.79
GRPO	72	77.42 ± 0.59	82.11 ± 1.76	**45.93 ± 1.60**	57.20 ± 1.40	**10.58 ± 1.95**	**20.49 ± 2.17**
MURPHY- MARS	72	**79.67 ± 3.01**	**86.58 ± 1.06**	44.73 ± 0.50	**62.00 ± 1.91**	6.98 ± 3.72	**20.25 ± 3.07**
Qwen3-4B							
Base	–	90.04 ± 3.13	93.49 ± 0.93	52.13 ± 0.42	70.93 ± 1.01	17.56 ± 1.78	36.03 ± 3.05
GRPO	72	88.61 ± 0.93	94.71 ± 0.93	51.73 ± 0.23	72.87 ± 0.90	20.95 ± 0.67	39.64 ± 2.73
MURPHY- MARS	72	**92.48 ± 0.61**	**95.73 ± 0.31**	**53.33 ± 1.15**	**73.33 ± 1.31**	**22.52 ± 2.47**	**41.44 ± 1.09**
OLMo-2-1124-7B-Instruct							
Base	–	37.20 ± 0.86	46.04 ± 0.43	19.90 ± 0.42	29.60 ± 0.28	1.35 ± 0.00	**3.72 ± 0.48**
GRPO	72	41.26 ± 0.35	43.70 ± 0.93	**32.87 ± 0.23**	35.80 ± 1.00	0.68 ± 0.00	2.70 ± 0.68
MURPHY- MARS	72	**45.53 ± 0.70**	**52.24 ± 1.96**	29.33 ± 0.90	**39.67 ± 1.29**	**1.80 ± 0.39**	3.38 ± 1.17

Table 1: Performance of Qwen3-1.7B, Qwen3-4B, and OLMo-2-1124-7B-Instruct variants across evaluation benchmarks. The *Rollouts* column indicates the total number of generations across both stages. MURPHY achieves competitive (single-turn; iter-1) or superior results (multi-turn; iter-3), with gains of up to 8% over GRPO under an equivalent compute budget.

Model & Strategy	Rollouts	HumanEval (%)		MBPP (%)		BigCodeBench (%)	
		Iter-1	Iter-3	Iter-1	Iter-3	Iter-1	Iter-3
Qwen3-1.7B							
MARS	72	**79.67 ± 3.01**	**86.58 ± 1.06**	44.73 ± 0.50	**62.00 ± 1.91**	6.98 ± 3.72	20.25 ± 3.07
MERS ($\gamma = 0.9$)	72	78.66 ± 2.11	84.76 ± 1.22	**46.53 ± 0.81**	60.53 ± 1.79	**10.81 ± 0.00**	**22.52 ± 5.67**
MERS ($\gamma = 1$)	72	78.46 ± 0.70	85.57 ± 0.35	45.07 ± 1.10	60.93 ± 1.29	9.46 ± 2.44	20.27 ± 1.35
OLMo-2-1124-7B-Instruct							
MARS	72	**45.53 ± 0.70**	**52.24 ± 1.96**	**29.33 ± 0.90**	**39.67 ± 1.29**	**1.80 ± 0.39**	**3.38 ± 1.17**
MERS ($\gamma = 1$)	72	37.40 ± 0.35	41.87 ± 1.27	27.87 ± 0.12	34.40 ± 2.40	0.90 ± 0.39	1.13 ± 0.39

Table 2: Ablation study comparing **Max Reward (MARS)** vs **Mean Reward (MERS)** propagation strategies across Qwen3-1.7B and OLMo-2-1124-7B-Instruct. MARS consistently outperforms MERS in both single (Iter-1) and multi-turn (Iter-3).

updates (denoted as *Updates* in Tab. 3). Here, we compare the effectiveness of these strategies and observe that INTERP consistently matches or surpasses the performance of MURPHY without pruning. This indicates that pruning across groups offers a more robust mechanism for eliminating redundant rollouts while retaining high-quality candidates. The ablation study is conducted using the Qwen3-1.7B model, and the results are reported in Tab. 3.

6 Conclusion

We introduced MURPHY, a multi-turn reflective reinforcement learning framework that extends RLVR algorithms by incorporating iterative self-correction through both quantitative and qualitative feedback. By grounding optimization in intermediate feedback signals and propagating rewards across refinement turns, MURPHY consistently enhances reasoning and code generation performance over GRPO, particularly in multi-iteration settings where feedback-driven refinement is crucial. These findings underscore the value of integrating structured feedback directly into the optimization

Model & Strategy	Updates	HumanEval (%)		MBPP (%)		BigCodeBench (%)	
		Iter-1	Iter-3	Iter-1	Iter-3	Iter-1	Iter-3
MURPHY (MARS)	72	79.67 ± 3.01	**86.58 ± 1.06**	44.73 ± 0.50	**62.00 ± 1.91**	6.98 ± 3.72	20.25 ± 3.07
MURPHY (MARS) – INTRAP	36	**82.34 ± 1.03**	84.28 ± 2.01	**45.73 ± 0.50**	59.87 ± 1.70	**11.03 ± 2.81**	21.17 ± 2.07
MURPHY (MARS) – INTERP	40	77.43 ± 2.20	86.17 ± 0.70	44.53 ± 0.90	61.20 ± 1.39	10.58 ± 2.56	**24.54 ± 2.82**

Table 3: Comparison of MURPHY and its pruned variants on evaluation benchmarks using Qwen3-1.7B. All variants generate 72 rollouts per query. The *Updates* column denotes the total number of gradient steps per query. Pruned variants (INTRAP, INTERP) maintain competitive performance relative to the non-pruned version with significant computation reduction.

process. Future directions include developing adaptive rollout selection strategies, extending MURPHY to broader reasoning domains, and coupling it with inference-time search to further strengthen both training- and test-time reasoning.

7 Limitations

While effective, MURPHY's multi-turn design increases computational cost; pruning partially mitigates this but it remains more resource-intensive than single-stage baselines. Our experiments are limited to structured feedback in code generation, leaving open questions about generalization to noisier feedback, deeper refinement chains, and broader notions of agentic performance such as robustness and alignment with developer intent.

8 Acknowledgments

We acknowledge that Fig. 2 incorporates icons sourced from Flaticon. Full creator attributions are provided in App. G.

References

[1] Arash Ahmadian, Chris Cremer, Matthias Gallé, Marzieh Fadaee, Julia Kreutzer, Olivier Pietquin, Ahmet Üstün, and Sara Hooker. Back to basics: Revisiting REINFORCE-style optimization for learning from human feedback in LLMs. In Lun-Wei Ku, Andre Martins, and Vivek 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, August 2024. doi: 10.18653/v1/2024.acl-long.662. URL https://aclanthology.org/2024.acl-long.662/.

[2] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2):235–256, 2002.

[3] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021. URL https://arxiv.org/abs/2108.07732.

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

[5] Mingyang Chen, Linzhuang Sun, Tianpeng Li, Haoze Sun, Yijie Zhou, Chenzheng Zhu, Haofen Wang, Jeff Z. Pan, Wen Zhang, Huajun Chen, Fan Yang, Zenan Zhou, and Weipeng Chen. Research: Learning to reason with search for llms via reinforcement learning, 2025. URL https://arxiv.org/abs/2503.19470.

[6] DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan

Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL https://arxiv.org/abs/2501.12948.

[7] Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. RLEF: Grounding code LLMs in execution feedback with reinforcement learning. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=PzSG5nKe1q.

[8] Arnav Kumar Jain, Gonzalo Gonzalez-Pumariega, Wayne Chen, Alexander M Rush, Wenting Zhao, and Sanjiban Choudhury. Multi-turn code generation through single-step rewards. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=aJeLhLcsh0.

[9] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *ACM Trans. Softw. Eng. Methodol.*, July 2025. ISSN 1049-331X. doi: 10.1145/3747588. URL https://doi.org/10.1145/3747588.

[10] Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan O Arik, Dong Wang, Hamed Zamani, and Jiawei Han. Search-r1: Training LLMs to reason and leverage search engines with reinforcement learning. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=Rwhi91ideu.

[11] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention, 2023. URL https://arxiv.org/abs/2309.06180.

[12] Vijay Lingam, Behrooz Omidvar Tehrani, Sujay Sanghavi, Gaurav Gupta, Sayan Ghosh, Linbo Liu, Jun Huan, and Anoop Deoras. Enhancing language model agents using diversity of thoughts. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=ZsP3YbYeE9.

[13] Samuel Miserendino, Michele Wang, Tejal Patwardhan, and Johannes Heidecke. SWE-lancer: Can frontier LLMs earn $1 million from real-world freelance software engineering? In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=xZXhFg43EI.

[14] Team OLMo, Pete Walsh, Luca Soldaini, Dirk Groeneveld, Kyle Lo, Shane Arora, Akshita Bhagia, Yuling Gu, Shengyi Huang, Matt Jordan, Nathan Lambert, Dustin Schwenk, Oyvind Tafjord, Taira Anderson, David Atkinson, Faeze Brahman, Christopher Clark, Pradeep Dasigi, Nouha Dziri, Michal Guerquin, Hamish Ivison, Pang Wei Koh, Jiacheng Liu, Saumya Malik, William Merrill, Lester James V. Miranda, Jacob Morrison, Tyler Murray, Crystal Nam, Valentina Pyatkin, Aman Rangapur, Michael Schmitz, Sam Skjonsberg, David Wadden, Christopher Wilhelm, Michael Wilson, Luke Zettlemoyer, Ali Farhadi, Noah A. Smith, and Hannaneh Hajishirzi. 2 olmo 2 furious, 2025. URL https://arxiv.org/abs/2501.00656.

[15] OpenAI, :, Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, Alex Iftimie, Alex Karpenko, Alex Tachard Passos, Alexander Neitz, Alexander Prokofiev, Alexander Wei, Allison Tam, Ally Bennett, Ananya Kumar, Andre Saraiva, Andrea Vallone, Andrew Duberstein, Andrew Kondrich, Andrey Mishchenko, Andy Applebaum, Angela Jiang, Ashvin Nair, Barret Zoph, Behrooz Ghorbani, Ben Rossen, Benjamin Sokolowsky, Boaz Barak, Bob McGrew, Borys Minaiev, Botao Hao, Bowen Baker, Brandon Houghton, Brandon McKinzie, Brydon Eastman, Camillo Lugaresi, Cary Bassin, Cary Hudson, Chak Ming Li, Charles de Bourcy, Chelsea Voss, Chen Shen, Chong Zhang, Chris Koch, Chris Orsinger, Christopher Hesse, Claudia Fischer, Clive Chan, Dan Roberts, Daniel Kappler, Daniel Levy, Daniel Selsam, David Dohan, David Farhi, David Mely, David Robinson, Dimitris Tsipras, Doug Li, Dragos Oprica, Eben Freeman, Eddie Zhang, Edmund Wong, Elizabeth Proehl, Enoch Cheung, Eric Mitchell, Eric Wallace, Erik Ritter, Evan Mays, Fan Wang, Felipe Petroski Such, Filippo Raso, Florencia Leoni, Foivos Tsimpourlas, Francis Song, Fred von Lohmann, Freddie Sulit, Geoff Salmon,

Giambattista Parascandolo, Gildas Chabot, Grace Zhao, Greg Brockman, Guillaume Leclerc, Hadi Salman, Haiming Bao, Hao Sheng, Hart Andrin, Hessam Bagherinezhad, Hongyu Ren, Hunter Lightman, Hyung Won Chung, Ian Kivlichan, Ian O'Connell, Ian Osband, Ignasi Clavera Gilaberte, Ilge Akkaya, Ilya Kostrikov, Ilya Sutskever, Irina Kofman, Jakub Pachocki, James Lennon, Jason Wei, Jean Harb, Jerry Twore, Jiacheng Feng, Jiahui Yu, Jiayi Weng, Jie Tang, Jieqi Yu, Joaquin Quiñonero Candela, Joe Palermo, Joel Parish, Johannes Heidecke, John Hallman, John Rizzo, Jonathan Gordon, Jonathan Uesato, Jonathan Ward, Joost Huizinga, Julie Wang, Kai Chen, Kai Xiao, Karan Singhal, Karina Nguyen, Karl Cobbe, Katy Shi, Kayla Wood, Kendra Rimbach, Keren Gu-Lemberg, Kevin Liu, Kevin Lu, Kevin Stone, Kevin Yu, Lama Ahmad, Lauren Yang, Leo Liu, Leon Maksin, Leyton Ho, Liam Fedus, Lilian Weng, Linden Li, Lindsay McCallum, Lindsey Held, Lorenz Kuhn, Lukas Kondraciuk, Lukasz Kaiser, Luke Metz, Madelaine Boyd, Maja Trebacz, Manas Joglekar, Mark Chen, Marko Tintor, Mason Meyer, Matt Jones, Matt Kaufer, Max Schwarzer, Meghan Shah, Mehmet Yatbaz, Melody Y. Guan, Mengyuan Xu, Mengyuan Yan, Mia Glaese, Mianna Chen, Michael Lampe, Michael Malek, Michele Wang, Michelle Fradin, Mike McClay, Mikhail Pavlov, Miles Wang, Mingxuan Wang, Mira Murati, Mo Bavarian, Mostafa Rohaninejad, Nat McAleese, Neil Chowdhury, Neil Chowdhury, Nick Ryder, Nikolas Tezak, Noam Brown, Ofir Nachum, Oleg Boiko, Oleg Murk, Olivia Watkins, Patrick Chao, Paul Ashbourne, Pavel Izmailov, Peter Zhokhov, Rachel Dias, Rahul Arora, Randall Lin, Rapha Gontijo Lopes, Raz Gaon, Reah Miyara, Reimar Leike, Renny Hwang, Rhythm Garg, Robin Brown, Roshan James, Rui Shu, Ryan Cheu, Ryan Greene, Saachi Jain, Sam Altman, Sam Toizer, Sam Toyer, Samuel Miserendino, Sandhini Agarwal, Santiago Hernandez, Sasha Baker, Scott McKinney, Scottie Yan, Shengjia Zhao, Shengli Hu, Shibani Santurkar, Shraman Ray Chaudhuri, Shuyuan Zhang, Siyuan Fu, Spencer Papay, Steph Lin, Suchir Balaji, Suvansh Sanjeev, Szymon Sidor, Tal Broda, Aidan Clark, Tao Wang, Taylor Gordon, Ted Sanders, Tejal Patwardhan, Thibault Sottiaux, Thomas Degry, Thomas Dimson, Tianhao Zheng, Timur Garipov, Tom Stasi, Trapit Bansal, Trevor Creech, Troy Peterson, Tyna Eloundou, Valerie Qi, Vineet Kosaraju, Vinnie Monaco, Vitchyr Pong, Vlad Fomenko, Weiyi Zheng, Wenda Zhou, Wes McCabe, Wojciech Zaremba, Yann Dubois, Yinghai Lu, Yining Chen, Young Cha, Yu Bai, Yuchen He, Yuchen Zhang, Yunyun Wang, Zheng Shao, and Zhuohan Li. Openai o1 system card, 2024. URL https://arxiv.org/abs/2412.16720.

[16] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. URL https://arxiv.org/abs/1707.06347.

[17] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, Y.K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.

[18] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu 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. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/1b44b878bb782e6954cd888628510e90-Paper-Conference.pdf.

[19] Kimi Team, Angang Du, Bofei Gao, Bowei Xing, Changjiu Jiang, Cheng Chen, Cheng Li, Chenjun Xiao, Chenzhuang Du, Chonghua Liao, Chuning Tang, Congcong Wang, Dehao Zhang, Enming Yuan, Enzhe Lu, Fengxiang Tang, Flood Sung, Guangda Wei, Guokun Lai, Haiqing Guo, Han Zhu, Hao Ding, Hao Hu, Hao Yang, Hao Zhang, Haotian Yao, Haotian Zhao, Haoyu Lu, Haoze Li, Haozhen Yu, Hongcheng Gao, Huabin Zheng, Huan Yuan, Jia Chen, Jianhang Guo, Jianlin Su, Jianzhou Wang, Jie Zhao, Jin Zhang, Jingyuan Liu, Junjie Yan, Junyan Wu, Lidong Shi, Ling Ye, Longhui Yu, Mengnan Dong, Neo Zhang, Ningchen Ma, Qiwei Pan, Qucheng Gong, Shaowei Liu, Shengling Ma, Shupeng Wei, Sihan Cao, Siying Huang, Tao Jiang, Weihao Gao, Weimin Xiong, Weiran He, Weixiao Huang, Weixin Xu, Wenhao Wu, Wenyang He, Xianghui Wei, Xianqing Jia, Xingzhe Wu, Xinran Xu, Xinxing Zu, Xinyu Zhou, Xuehai Pan, Y. Charles, Yang Li, Yangyang Hu, Yangyang Liu, Yanru Chen, Yejie Wang, Yibo Liu, Yidao Qin, Yifeng Liu, Ying Yang, Yiping Bao, Yulun Du, Yuxin Wu, Yuzhi Wang, Zaida Zhou, Zhaoji Wang, Zhaowei Li, Zhen Zhu, Zheng Zhang, Zhexu Wang, Zhilin Yang, Zhiqi Huang, Zihao Huang, Ziyao Xu, Zonghan Yang, and Zongyu Lin. Kimi k1.5: Scaling reinforcement learning with llms, 2025. URL https://arxiv.org/abs/2501.12599.

[20] Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, Shengyi Huang, Kashif Rasul, and Quentin Gallouédec. TRL: Transformer Reinforcement Learning. https://github.com/huggingface/trl, 2020.

[21] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying llm-based software engineering agents. *Proc. ACM Softw. Eng.*, 2(FSE), June 2025. doi: 10.1145/3715754. URL https://doi.org/10.1145/3715754.

[22] Yixuan Even Xu, Yash Savani, Fei Fang, and Zico Kolter. Not all rollouts are useful: Down-sampling rollouts in llm reinforcement learning. *arXiv preprint arXiv:2504.13818*, 2025.

[23] Zhangchen Xu, Yang Liu, Yueqin Yin, Mingyuan Zhou, and Radha Poovendran. Kodcode: A diverse, challenging, and verifiable synthetic dataset for coding. *arXiv preprint arXiv:2503.02951*, 2025.

[24] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

[25] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=mXpq6ut8J3.

[26] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

[27] Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Mu Qiao, Yonghui Wu, and Mingxuan Wang. Dapo: An open-source llm reinforcement learning system at scale, 2025. URL https://arxiv.org/abs/2503.14476.

[28] Yufeng Yuan, Yu Yue, Ruofei Zhu, Tiantian Fan, and Lin Yan. What's behind ppo's collapse in long-cot? value optimization holds the secret, 2025. URL https://arxiv.org/abs/2503.01491.

[29] Yu Yue, Yufeng Yuan, Qiying Yu, Xiaochen Zuo, Ruofei Zhu, Wenyuan Xu, Jiaze Chen, Chengyi Wang, TianTian Fan, Zhengyin Du, Xiangpeng Wei, Xiangyu Yu, Gaohong Liu, Juncai Liu, Lingjun Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, Ru Zhang, Xin Liu, Mingxuan Wang, Yonghui Wu, and Lin Yan. Vapo: Efficient and reliable reinforcement learning for advanced reasoning tasks, 2025. URL https://arxiv.org/abs/2504.05118.

[30] Shaokun Zhang, Yi Dong, Jieyu Zhang, Jan Kautz, Bryan Catanzaro, Andrew Tao, Qingyun Wu, Zhiding Yu, and Guilin Liu. Nemotron-research-tool-n1: Tool-using language models with reinforced reasoning. *arXiv preprint arXiv:2505.00024*, 2025.

[31] Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong Liu, Rui Men, An Yang, Jingren Zhou, and Junyang Lin. Group sequence policy optimization, 2025. URL https://arxiv.org/abs/2507.18071.

[32] Li Zhong, Zilong Wang, and Jingbo 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.

[33] Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning, acting, and planning in language models. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=njwv9BsGHF.

[34] Richard Zhuang*, Trung Vu*, Alex Dimakis, and Maheswaran Sathiamoorthy. Improving multi-turn tool use with reinforcement learning. https://www.bespokelabs.ai/blog/improving-multi-turn-tool-use-with-reinforcement-learning, 2025. Accessed: 2025-04-17.

[35] Terry Yue Zhuo, Vu Minh Chien, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen GONG, James Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, David Lo, Binyuan Hui, Niklas Muennighoff, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro Von Werra. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=YrycTjllL0.

Appendix

Contents

A Extended Related Work

LLM Agents for Software Development. Recent works [9, 32] 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 [18, 12], which leverage execution feedback to generate self-reflections for refining candidate programs [25, 21]. 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 [17] revived interest in RL as an efficient alternative to PPO [16], offering comparable reasoning performance with far lower computational cost. Subsequent variants [29, 27, 28, 31] 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 [5, 10], tool use [34, 30], and code generation [8, 7]. 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 [8] and RLEF [7], 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.

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 $\{o_{q,1}, o_{q,2}, \dots, o_{q,G}\}$. 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 $\mathbf{r}_q = \{r_{q,1}, r_{q,2}, \dots, r_{q,G}\}$. 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, $D_{\text{KL}}(\pi_\theta || \pi_{\theta_{\text{old}}})$ 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. 2.

Definition 2. (GRPO Objective)

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

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 Reflexion

Reflexion [18] 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.

D Implementation Details

We implement our framework on top of TRL [20], 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. E. All experiments use publicly available datasets. The base models (Qwen3, OLMo) are available for research use under the Apache 2.0 license.

D.1 Model Size and Compute Budget

All experiments were conducted on $8\times$ NVIDIA H100 GPUs. Our implementation builds on HuggingFace's TRL[4]. For efficiency, 2 GPUs were allocated for inference via vLLM, while the remaining 6 GPUs handled model updates. Training Qwen3-1.7B with MURPHY on 1,000 KodCode samples took approximately 1.5 hours, Qwen3-4B took 4 hours, and OLMo-2-7B-Instruct required 10 - 13 hours. Checkpoints were saved every 50 steps, and for all baselines, we selected the checkpoint corresponding to one epoch.

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

D.2 Hyperparameters

We set the KL regularization factor $\beta = 0.04$, learning rate to 10^{-6}, and weight decay to 0.1 for both GRPO and MURPHY variants. Unless stated otherwise, the number of stages in MURPHY is set to 2. For GRPO and the first stage of MURPHY, we use 8 rollouts per prompt, while the second stage uses up to 8 rollouts per prompt (resulting in a maximum of 64 rollouts). To ensure a fair computational comparison, we also train GRPO with 72 rollouts. Following Yang et al., we set the temperature to 0.6 and top-p to 0.95 for all experiments.

D.3 Package Parameters

We use the Reflexion [18] 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
```

D.4 Additional Experiments

D.4.1 Ablation: Comparison with a Naive Multi-Turn Extension

A straightforward way to adapt GRPO for multi-turn training is to extend failed rollouts by appending feedback from previous turns without introducing additional fan-out. Rewards and advantages are then computed on the final turn, and the GRPO objective is applied. We compare this baseline against MURPHY in Tab. 4. The number of turns is set to 2. The results underscore the importance of MURPHY's structured credit assignment and staged fan-out, both of which contribute to its superior multi-turn optimization performance.

D.4.2 Ablation: Effect of Training Dataset Size

To examine the effect of training dataset size, we construct three nested subsets of KodCode (2K $\subset$ 3K $\subset$ 4K) and train Qwen3-1.7B using GRPO (72 rollouts) and MURPHY. Results are summarized in Tab. 5. While performance does not increase monotonically with dataset size, MURPHY consistently demonstrates superior multi-turn robustness and maintains an average improvement of up to $\sim 9\%$ over GRPO across all dataset scales.

D.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. 6 show that the pruned variant achieves competitive or even superior performance compared to the non-pruned counterpart.

Model	Rollouts	HumanEval (%)		MBPP (%)		BigCodeBench (%)	
		Iter-1	Iter-3	Iter-1	Iter-3	Iter-1	Iter-3
Qwen3-1.7B							
Base	–	74.19 ± 1.95	80.07 ± 1.27	43.47 ± 0.64	53.93 ± 3.52	7.20 ± 2.73	18.24 ± 1.79
GRPO	72	$\underline{77.42 \pm 0.59}$	82.11 ± 1.76	$\underline{45.93 \pm 1.60}$	57.20 ± 1.40	$\underline{10.58 \pm 1.95}$	$\mathbf{20.49 \pm 2.17}$
MURPHY (Simple)	144	70.33 ± 0.93	$\underline{82.11 \pm 0.35}$	$\mathbf{46.00 \pm 0.40}$	$\underline{58.53 \pm 1.03}$	$\mathbf{13.29 \pm 1.41}$	18.24 ± 1.17
MURPHY- MARS	72	$\mathbf{79.67 \pm 3.01}$	$\mathbf{86.58 \pm 1.06}$	44.73 ± 0.50	$\mathbf{62.00 \pm 1.91}$	6.98 ± 3.72	$\underline{20.25 \pm 3.07}$
OLMo-2-1124-7B-Instruct							
Base	–	37.20 ± 0.86	$\underline{46.04 \pm 0.43}$	19.90 ± 0.42	29.60 ± 0.28	$\underline{1.35} \pm 0.00$	$\mathbf{3.72 \pm 0.48}$
GRPO	72	$\underline{41.26 \pm 0.35}$	43.70 ± 0.93	$\mathbf{32.87 \pm 0.23}$	$\underline{35.80 \pm 1.00}$	0.68 ± 0.00	2.70 ± 0.68
MURPHY (Simple)	144	39.02 ± 0.00	41.87 ± 0.93	28.33 ± 0.31	33.60 ± 0.69	1.13 ± 1.41	2.25 ± 0.39
MURPHY- MARS	72	$\mathbf{45.53 \pm 0.70}$	$\mathbf{52.24 \pm 1.96}$	$\underline{29.33 \pm 0.90}$	$\mathbf{39.67 \pm 1.29}$	$\mathbf{1.80 \pm 0.39}$	$\underline{3.38 \pm 1.17}$

Table 4: Performance of Qwen3-1.7B and OLMo-2-1124-7B-Instruct variants on HumanEval, MBPP, and BigcodeBench benchmarks, reported as pass@1 accuracy (% mean ± stdev over 3 independent runs). Best results are **bold**, second-best are underlined. MURPHY- MARS outperforms MURPHY (Simple) on average.

Qwen3-1.7B	Dataset	HumanEval Iter-1	HumanEval Iter-3	MBPP Iter-1	MBPP Iter-3	BigcodeBench Iter-1	BigcodeBench Iter-3
MURPHY- MARS	2K	$\mathbf{81.65\% \pm 2.19}$	$\mathbf{84.15\% \pm 2.05}$	$\mathbf{46.93\% \pm 0.81}$	$\mathbf{60.47\% \pm 2.66}$	$\mathbf{2.70\% \pm 0.68}$	$\mathbf{11.04\% \pm 1.70}$
GRPO	2K	$78.46\% \pm 2.46$	$83.54\% \pm 1.06$	$46.13\% \pm 0.70$	$57.67\% \pm 0.61$	$2.48\% \pm 0.78$	$8.11\% \pm 1.17$
MURPHY- MARS	3K	$77.30\% \pm 3.55$	$\mathbf{84.55\% \pm 1.96}$	$45.53\% \pm 0.12$	$\mathbf{63.60\% \pm 0.72}$	$\mathbf{3.83\% \pm 0.78}$	$\mathbf{10.36\% \pm 1.41}$
GRPO	3K	$\mathbf{79.07\% \pm 2.54}$	$80.49\% \pm 1.61$	$\mathbf{47.40\% \pm 1.20}$	$54.87\% \pm 1.68$	$3.60\% \pm 1.03$	$6.08\% \pm 0.68$
MURPHY- MARS	4K	$79.27\% \pm 2.26$	$\mathbf{89.63\% \pm 1.22}$	$45.60\% \pm 1.31$	$\mathbf{62.80\% \pm 0.92}$	$\mathbf{3.60\% \pm 0.39}$	$\mathbf{8.78\% \pm 2.70}$
GRPO	4K	$\mathbf{79.88\% \pm 2.66}$	$83.33\% \pm 0.70$	$\mathbf{47.27\% \pm 0.46}$	$57.40\% \pm 1.83$	$2.70\% \pm 1.17$	$6.98\% \pm 1.41$

Table 5: Ablation: average ± stdev (percent) for MURPHY vs GRPO on KodCode (2K/3K/4K). MURPHY MARS shows gains of up to $\sim 9\%$ over compute equivalent GRPO.

Model	HumanEval		MBPP		BigCodeBench	
	Iter-1	Iter-3	Iter-1	Iter-3	Iter-1	Iter-3
GRPO	77.24 ± 2.14	83.13 ± 1.96	44.07 ± 0.31	53.93 ± 0.61	1.13 ± 0.39	6.53 ± 0.78
MURPHY- MARS, INTERP	78.66 ± 2.66	84.15 ± 1.83	45.53 ± 1.17	61.20 ± 0.53	2.70 ± 0.68	7.43 ± 1.79
MURPHY- MARS	79.27 ± 2.11	83.94 ± 1.41	44.80 ± 1.11	62.00 ± 1.40	1.35 ± 0.68	5.86 ± 0.39

Table 6: 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 or superior performance compared to non-pruned MURPHY.

E 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.

F Potential Risks

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.

G Icon Attributions

The icons used in Fig. 2 were obtained from Flaticon and are attributed to the following creators: zero_wing (Prompt), Freepik (AI Model), Smashicons (Code Snippet), juicy_fish (Code Executor), Freepik (Doc), kliwir art (Correct), and Rakib Hassan Rahim (Wrong).