Extended Abstract

Motivation. RL fine-tuning with verifiable rewards is a central recipe for improving the reasoning of small language models, and the Countdown game—combining a few numbers with arithmetic to hit a target—is a standard, cheaply verifiable testbed. The difficulty is that a model's natural-language chain-of-thought is only loosely coupled to the answer it commits to: it can produce fluent reasoning while making hidden arithmetic or number-usage errors, and a sparse final-answer reward gives no signal about *where* the reasoning went wrong. We ask whether replacing free-form reasoning with a restricted, *executable* program turns this sparse reward into a graded, more interpretable training signal, using a standard SFT→RLOO pipeline as the reference point.

Method. The baseline supervised fine-tunes (SFT) Qwen2.5-0.5B on Countdown and then aligns it with on-policy RLOO (REINFORCE Leave-One-Out) Ahmadian et al. (2024), using a leave-one-out group baseline, per-sample importance weighting to correct the vLLM-sampler/Hugging-Face-policy mismatch, and entropy and token-level KL regularization. Our extension, *programmatic RLOO*, keeps this loop and replaces only the reward: the prompt is rewritten to forbid free-form reasoning and request a restricted Python-style program—one assignment per line of the form `variable = operand operator operand`, ending in a final `answer`. A verifier parses the program with Python's `ast` under a four-operator whitelist, evaluates it while tracking which original numbers each value consumes, and emits a *decomposed* reward $R = 0.05\, r_{\text{format}} + 0.10\, r_{\text{parse}} + 0.15\, r_{\text{exec}} + 0.20\, r_{\text{num-use}} + 0.50\, r_{\text{target}}$. Unlike PAL/PoT (programs at prompting time, computed by an interpreter) and RLEF (interactive code-execution feedback), our program is a single static artifact verified *after* generation purely to supply a decomposed, interpretable RLOO reward—this post-hoc, component-wise verification of a non-code reasoning task is the project's novelty.

Implementation. Both runs use Qwen2.5-0.5B on `countdown_tasks_3to4`, initialized from the provided SFT checkpoint, trained for 100 RLOO iterations on a single H100 (Modal) with $k = 16$ rollouts per prompt, AdamW at 1×10^{-5}, and entropy/KL coefficients of 0.001. The programmatic run is identical except for the reward (and prompt): the parser exposes no builtins, and each assignment updates a variable table plus the multiset of original numbers it consumes, from which the five components are scored.

Results. On the 50-problem test set, SFT reaches pass@1 0.338; standard RLOO lifts this to 0.551 with stable dynamics (KL grows smoothly to 0.125, importance weight stays in $[0.58, 1.33]$, entropy falls $0.43 \to 0.19$). Programmatic RLOO pushes pass@1 highest, to 0.613—a 6.1 percentage-point gain over standard RLOO—with near-perfect program validity (parse/execution ≈ 0.996) and 0.93 legal number usage. This advantage is concentrated at the first sample: by pass@2 the programmatic curve is essentially tied with standard RLOO, and for larger k it falls below it. The trade-off is diversity: pass@16 is 0.80 for SFT, 0.78 for standard RLOO, and 0.72 for programmatic RLOO, so the pass@k curves cross.

Discussion & Conclusion. Execution-based programmatic reward is an effective way to buy single-sample accuracy and interpretability on Countdown: it attributes every lost point to a specific stage of the computation, which a sparse final-answer reward folds into an undifferentiated zero. But it sharpens rather than resolves RL's core trade-off. The shaped reward concentrates probability mass on a smaller set of valid, high-reward programs, so the model's most likely completion improves, while the additional samples become less diverse and less likely to cover alternative correct solution paths. The most promising next step is to pair the decomposed programmatic reward with a diversity-preserving objective so that pass@1 gains no longer come at the expense of pass@k.

Programmatic Reasoning for Countdown: Learning to Generate Executable Python-Style Verifications

Oleh Ivankiv
Department of Computer Science
Stanford University
ivankiv@stanford.edu

Henry Zhou
Department of Computer Science
Stanford University
henryjz@stanford.edu

Abstract

We study reinforcement-learning fine-tuning of a small language model (Qwen2.5-0.5B) on the Countdown arithmetic-reasoning task, where the model must combine given numbers with $+, -, \times, \div$ to reach a target. Free-form chain-of-thought is hard to verify and a sparse final-answer reward provides little signal about where reasoning fails. We first establish and carefully analyze a baseline pipeline—supervised fine-tuning followed by on-policy RLOO with a leave-one-out baseline, importance-weighted to correct for the mismatch between the vLLM sampler and the Hugging Face policy, and regularized with entropy and a token-level KL penalty. This baseline raises the mean training rollout reward from 0.296 to 0.611 with stable, well-behaved training dynamics, and improves test-set pass@1 from ≈ 0.33 to ≈ 0.55. Replacing the binary verifier with a restricted Python-style execution reward raises single-sample accuracy further (pass@1 0.613) and yields near-perfect program validity, isolating the remaining errors to semantic mistakes (wrong target, illegal number reuse). The gain is concentrated at pass@1: programmatic RLOO improves the model's most likely response but falls below standard RLOO at larger k, showing that decomposed execution feedback trades output diversity for sharper, more diagnosable single-sample reasoning.

1 Introduction

Reinforcement learning with verifiable rewards has become a standard tool for improving the reasoning of language models: rather than imitating human demonstrations, a model is optimized directly against an automatic checker that scores its outputs Ahmadian et al. (2024); DeepSeek-AI (2025). The Countdown game is a widely used benchmark for this paradigm because it is trivially verifiable yet genuinely hard: given a small set of integers and a target, the model must produce an arithmetic expression that uses each number at most once and evaluates exactly to the target Gandhi et al. (2024); Pan et al. (2025). Its large branching factor demands planning, search, and backtracking, while its verifier is a few lines of code, which makes it an ideal setting for studying RL fine-tuning at small scale.

The core difficulty that motivates this project is that the natural-language reasoning a model emits is only loosely tied to the answer it commits to. A model can generate confident, fluent reasoning that contains a hidden arithmetic slip or reuses a forbidden number, yet still be scored only on its final answer. A sparse terminal reward therefore tells the model *that* it failed but not *where*, which both slows credit assignment and makes failures hard to diagnose. One way to tighten the coupling between reasoning and answer is to make the reasoning itself executable, so that each intermediate step can be checked.

We organize the work around three research questions. **RQ1 (baseline):** On Countdown, how much does on-policy RLOO improve a supervised Qwen2.5-0.5B policy over SFT, and are its training

dynamics (reward, KL, importance weight, entropy) stable and well-behaved under an off-policy sampler? **RQ2 (extension):** Does replacing the binary verifier with a restricted, executable Python-style reward improve rollout accuracy and pass@k relative to standard free-form RLOO? **RQ3 (extension):** Does decomposing the reward into format, parse, execution, number-use, and target components yield a more interpretable training signal that localizes *where* a solution fails? This report develops RQ1 in full and reports the head-to-head comparison that answers RQ2 and the failure-mode diagnostics that answer RQ3.

2 Related Work

RL fine-tuning and preference optimization. Classical RLHF couples a learned reward model with PPO Ouyang et al. (2022); Schulman et al. (2017), which is effective but computationally heavy and sensitive to value estimation. Direct preference methods such as DPO Rafailov et al. (2023) and its theoretical generalization IPO Azar et al. (2024) bypass online sampling by optimizing a policy directly from pairwise preferences. In a separate line, Ahmadian et al. (2024) show that for LLM alignment many PPO components are unnecessary and that simple REINFORCE-style Williams (1992) updates with a leave-one-out baseline (RLOO) are competitive while being far cheaper; related group-relative methods such as GRPO underpin recent reasoning models DeepSeek-AI (2025). We adopt RLOO as our on-policy baseline because its leave-one-out baseline is parameter-free, it pairs naturally with a rule-based verifier reward, and it matches the small-model, single-GPU regime of this project.

RL for Countdown. Countdown was popularized as a reasoning testbed by Stream of Search Gandhi et al. (2024), which trains models on serialized search traces, and by TinyZero Pan et al. (2025), which reproduces R1-style RL on the task at very low cost. We build directly on this setting, using the same `Countdown-Tasks-3to4` family of problems and a rule-based verifier reward.

Executable and tool-augmented reasoning (extension). Our extension builds on work showing that language models benefit from executable or tool-augmented reasoning. Program-Aided Language Models (PAL) Gao et al. (2023) and Program of Thoughts (PoT) Chen et al. (2023) generate programs as intermediate reasoning and delegate exact computation to a Python runtime, separating reasoning from calculation. These are primarily prompting-time methods; we instead ask whether a restricted programmatic representation can improve the *reward signal* during RLOO fine-tuning. Toolformer Schick et al. (2023) and ToRA study broader tool-integrated reasoning, but typically rely on interactive tool calls or curated tool-use trajectories, whereas we generate a single restricted program verified after generation. Finally, RLEF Gehring et al. (2024) studies RL from execution feedback for code synthesis; our setting differs because Countdown is not a code-generation benchmark—code is introduced only as a structured abstraction for arithmetic reasoning. The novelty of our extension is to use restricted Python-style verification to supply decomposed, interpretable reward components—syntax validity, execution success, legal number usage, and final-target correctness—for RLOO.

3 Method

Base pipeline. We follow the standard two-stage recipe: supervised fine-tuning of Qwen2.5-0.5B on Countdown completions, then RL alignment of the SFT policy. The SFT stage minimizes token-level cross-entropy on the demonstrated `<think>/<answer>` completions and produces the checkpoint that serves as both the initial policy and the frozen reference for RL.

RLOO with a leave-one-out baseline. For each prompt x we draw k completions $y^{(1)}, \ldots, y^{(k)}$ and score each with the rule-based verifier $R(y, x)$. The leave-one-out advantage uses the mean of the *other* samples in the group as a per-sample baseline,

$$A_i \;=\; R(y^{(i)}, x) \;-\; \frac{1}{k-1} \sum_{j \neq i} R(y^{(j)}, x), \tag{1}$$

which is unbiased and parameter-free (no learned critic). The policy-gradient loss is taken over response tokens only.

Algorithm 1: RLOO fine-tuning for Countdown (base pipeline)

Input: SFT policy π_θ, frozen reference π_{ref}, verifier R, group size k
for *iteration* $= 1$ *to* T **do**
  Sample a batch of prompts $\{x\}$; for each x draw k rollouts $y^{(1)}, \ldots, y^{(k)} \sim \mu(\cdot \mid x)$ (vLLM);
  Score each rollout: $r_i \leftarrow R(y^{(i)}, x)$;
  Compute leave-one-out advantages $A_i \leftarrow r_i - \frac{1}{k-1} \sum_{j \neq i} r_j$;
  Recompute $\log \pi_\theta(y^{(i)} \mid x)$ over the response mask; set
   $w_i \leftarrow \exp(\text{clip}(\log \pi_\theta - \log \mu, \leq 2))$;
  $\mathcal{L} \leftarrow -\frac{1}{k} \sum_i w_i \, A_i \log \pi_\theta(y^{(i)} \mid x) - \beta_H \mathcal{H}[\pi_\theta] + \beta_{\text{KL}} \, \text{KL}(\pi_\theta \| \pi_{\text{ref}})$;
  Update θ with AdamW; clip gradient norm to 1.0;

Off-policy correction. Because rollouts are generated by a fast inference engine (vLLM) while the gradient step is taken in Hugging Face, the sampling distribution μ can drift from the current policy π_θ within an iteration. We therefore treat the sampler as a behavior policy and apply per-sample importance weighting

$$w(y, x) \;=\; \exp\big(\log \pi_\theta(y \mid x) - \log \mu(y \mid x)\big), \tag{2}$$

computed in log space with the log-ratio clipped at 2.0 for numerical stability. The resulting objective, with entropy regularization and a token-level KL penalty to the frozen SFT reference π_{ref}, is

$$\mathcal{L}(\theta) \;=\; -\mathbb{E}\big[\, w(y, x)\, A \, \log \pi_\theta(y \mid x) \,\big] \;-\; \beta_H \, \mathcal{H}[\pi_\theta] \;+\; \beta_{\text{KL}} \, \text{KL}\big(\pi_\theta \,\|\, \pi_{\text{ref}}\big). \tag{3}$$

Sequence log-probabilities are summed only over real response tokens: the prompt is masked out, and the right-padding introduced by the response tokenizer is excluded from the response mask so that $\log \pi_\theta(y \mid x)$ matches the quantity the sampler reports. This masking detail is necessary for the importance weights to be meaningful—without it the log-ratios are dominated by padding tokens.

Programmatic extension. The extension keeps Algorithm 1 unchanged and replaces only the reward R (and the prompt template). Instead of a free-form chain of thought ending in a single `<answer>` expression, the prompt instructs the model to emit a *restricted Python-style program* inside the answer tags: one assignment per line of the form `variable = operand operator operand`, where operands are original numbers or previously defined variables, operators are limited to $+, -, \times, \div$, and the program ends with a final variable (`answer`). The natural-language prefill ("Let me solve this step by step.") is removed so the first generated tokens can be code-like.

A verifier then scores the program with five checks. It extracts the last `<answer>` span and parses each non-empty line with Python's `ast` module, requiring a single assignment to a simple name. Each right-hand side is evaluated by a restricted interpreter that accepts only numeric constants, previously bound variables, the four binary operators, and unary $\pm$ (division by zero raises an error); no builtins are exposed. The interpreter simultaneously tracks, for every value, the multiset of original number literals that produced it. The final value is read from the variable named `answer` (or `final`/`result`, else the last assignment). The five checks are: *format* (well-formed answer tags), *parse* and *execution* (all lines parse and evaluate without error), *legal number use* (the multiset of consumed numbers exactly equals the provided numbers), and *target* (legal number use *and* final value within 10^{-5} of the target). They are combined into the decomposed reward

$$R \;=\; 0.05\, r_{\text{format}} + 0.10\, r_{\text{parse}} + 0.15\, r_{\text{exec}} + 0.20\, r_{\text{num-use}} + 0.50\, r_{\text{target}} \;\in\; [0, 1], \tag{4}$$

which replaces the binary verifier in Algorithm 1. The weights front-load most credit on hitting the target (0.50) while granting graded partial credit for a parseable, executable, legally-numbered program, so a model that cannot yet solve a problem still receives a graded signal toward valid intermediate structure. We make two assumptions: the model emits a single program per rollout (only the last answer span is scored), and verification is non-interactive (the program is checked after generation, with no tool calls during decoding). Relative to PAL/PoT, where a program's purpose is to *compute* the answer at inference time, here the program is an abstraction whose *decomposed verification* shapes the RL reward; relative to RLEF's interactive code-execution feedback, the signal is a single post-hoc, component-wise score.

4 Experimental Setup

Task and data. We use the Countdown 3-to-4 problem family (`asingh15/countdown_tasks_3to4`, derived from `Countdown-Tasks-3to4` Pan et al. (2025)), in which each problem provides three or four integers and a target, and a solution is an arithmetic expression that uses each number at most once and evaluates exactly to the target. Prompts request reasoning inside `<think>` tags and the final expression inside `<answer>` tags. The verifier awards full credit only when the expression is legal and hits the target, and a small format credit (0.1) when the output is well-formed but incorrect.

Base model and baselines. The base model is Qwen2.5-0.5B Qwen Team (2024). We compare three systems: (1) the **SFT** checkpoint (`asingh15/qwen-sft-countdown-defaultproj`); (2) **standard RLOO**, which fine-tunes the SFT policy against the binary rule-based verifier; and (3) **programmatic RLOO**, which uses the restricted execution reward. SFT is the natural lower bound (no RL signal) and standard RLOO isolates the effect of on-policy optimization with the existing reward, so that any further gain from the extension can be attributed to the reward shaping rather than to RL itself. All three systems share the same base checkpoint, sampler, and evaluation protocol, so differences are attributable to the training objective alone.

Metrics. We report *pass@*k (probability that at least one of k sampled completions solves the problem) on a held-out test set of 50 problems, computed by sampling 16 completions per prompt, scoring each with the verifier, and applying the unbiased pass@k estimator of Chen et al. (2021). On the test set the mean per-sample solve rate coincides with pass@1 by construction, so we report pass@1 and pass@16 as the headline functional metrics. For RL dynamics we track the policy-gradient loss, the mean importance weight, the token-level KL to the reference, and per-token entropy, since these diagnose whether the off-policy correction and KL penalty are keeping training stable; we additionally log the mean training rollout reward each iteration. For the programmatic variant we report the component diagnostics—program parse rate, execution-success rate, legal number-usage rate, and exact final-target rate—to isolate the effect of the decomposed reward and to localize failures.

Training details. SFT minimizes token cross-entropy on the demonstrated completions with AdamW (learning rate 5×10^{-6}, cosine schedule, warmup ratio 0.05, weight decay 0.01) for one epoch at batch size 16. RLOO trains for 100 outer iterations with AdamW (learning rate 1×10^{-5}, constant schedule; weight decay 1×10^{-4}; no warmup), an effective batch of 128 prompts with $k = 16$ rollouts per prompt (2048 trajectories per iteration), gradient accumulation over 128 microbatches (one prompt-group each), entropy coefficient 0.001, KL coefficient 0.001, gradient-norm clipping at 1.0, and importance-weight clipping at $\log w \leq 2.0$. Sampling uses temperature 1.0, top-$p = 1.0$, and top-k disabled. The programmatic run reuses this configuration verbatim and changes only the reward function. Training runs on a single H100 via Modal, with vLLM for sampling and Hugging Face for the gradient step. Key hyperparameters are summarized in Table 1.

5 Results

5.1 Quantitative Evaluation

SFT baseline. SFT converges quickly (Figure 1): training cross-entropy drops from ≈ 6 to below 1 within the first few steps and then plateaus, while train token accuracy rises to ≈ 0.90. Test cross-entropy and test token accuracy are essentially flat (a slight rise in test loss and a sub-0.001 drift in test token accuracy), indicating the supervised objective has saturated and is beginning to mildly overfit the surface form rather than improving held-out behavior. On the functional metric the SFT checkpoint reaches pass@1 ≈ 0.33 and pass@16 ≈ 0.80 (Figure 4). The large gap between pass@1 and pass@16 shows that the SFT model frequently *can* find a correct expression but does so unreliably on any single sample—exactly the regime where on-policy RL is expected to help.

RLOO dynamics. Standard RLOO trains stably and monotonically improves task success. The mean training rollout reward rises from 0.296 at step 0 to 0.611 at step 99—more than a $2\times$ improvement in actual task success (Figure 3, top-row accuracy panel). The optimization diagnostics

Table 1: Key training hyperparameters. The programmatic run is identical to standard RLOO except for the reward function.

	SFT	**RLOO (standard / programmatic)**
Base model	Qwen2.5-0.5B	Qwen2.5-0.5B (SFT init)
Optimizer	AdamW	AdamW
Learning rate	5×10^{-6} (cosine)	1×10^{-5} (constant)
Weight decay	1×10^{-2}	1×10^{-4}
Warmup ratio	0.05	0.0
Iterations	1 epoch	100 outer iters
Batch size	16	128 prompts
Rollouts / prompt (k)	—	16
Entropy coeff. β_H	—	0.001
KL coeff. β_{KL}	—	0.001
Grad-norm clip	1.0	1.0
Importance-weight clip	—	$\log w \leq 2.0$
Sampling temp / top-p	—	1.0 / 1.0

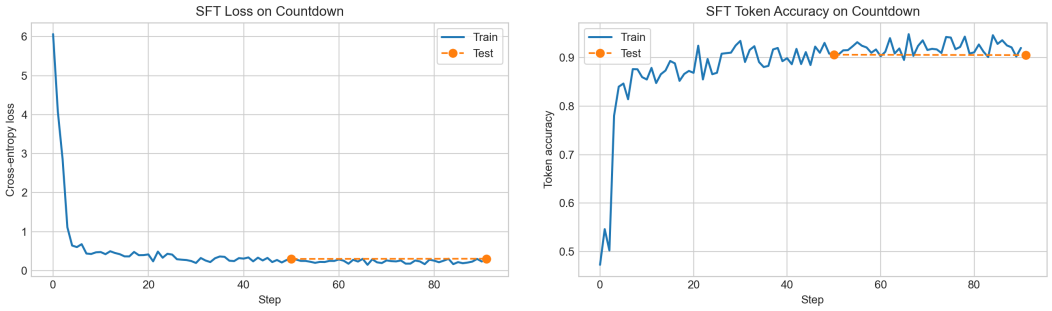

Figure 1: SFT training curves on Countdown. **Left:** token-level cross-entropy. **Right:** token accuracy. The train metric (line) is logged every step; the test metric (markers) is logged at steps 50 and 91. Train loss falls from ≈ 6 to < 1 and train token accuracy rises to ≈ 0.90 within a few steps, after which both plateau; the two test points sit right on the train curve, indicating the supervised objective has saturated with no meaningful held-out improvement.

confirm the run is healthy rather than collapsing: the token-level KL to the SFT reference grows smoothly and concavely from 0 to 0.125, so the 0.001 KL penalty is enough to prevent reward hacking without over-constraining the policy; per-token entropy decreases from 0.43 to 0.19 as the policy sharpens on the rewarded `<think>/<answer>` format; and the policy-gradient loss moves from -13.22 to $+2.20$. The off-policy correction never binds: the mean importance weight stays near 1.23 for the first $\sim$40 iterations and drifts down to 0.67 by step 99 as the policy diverges from the cached behavior policy, but it remains within $[0.58, 1.33]$ throughout and never approaches the clip threshold $e^2 \approx 7.4$. The mild drift toward $w < 1$ reflects the policy concentrating on trajectories the slightly stale sampler considered less likely, rather than any pathological divergence.

SFT vs. RLOO vs. programmatic on the test set. On held-out problems, both RL systems are markedly stronger than SFT at low k: standard RLOO lifts pass@1 from 0.338 (SFT) to 0.551, and the programmatic reward raises it further to 0.613. This is the clearest win for the extension: it improves the model's most likely sampled answer by 6.1 percentage points over standard RLOO. However, the advantage is almost entirely a pass@1 effect. At pass@2 the programmatic run (0.663) is already slightly below standard RLOO (0.665), and the gap widens as k grows; by $k = 16$, SFT (0.800) edges out standard RLOO (0.780), which in turn beats programmatic RLOO (0.720) (Figure 4). This pattern suggests that programmatic reward shaping improves reliability of the modal response but reduces coverage across samples. The verifier rewards well-formed, executable programs and legal number use, so training concentrates probability mass on a smaller set of structured solution templates. That concentration helps when only one answer is sampled, but it hurts pass@k at larger k, where success depends on diverse attempts that explore different arithmetic decompositions. The

headline comparison is collected in Table 2, and the component diagnostics for the programmatic run in Table 3.

Table 2: Functional correctness on the 50-problem Countdown test set (16 samples/prompt, unbiased pass@k). Best per column in **bold**.

Method	pass@1	pass@16
SFT baseline	0.338	**0.800**
Standard RLOO	0.551	0.780
Programmatic RLOO	**0.613**	0.720

Table 3: Programmatic-RLOO reward-component diagnostics, averaged over all test rollouts. Validity is near-perfect; the residual gap to a solved problem is almost entirely semantic (legal number use and hitting the target).

	answer tags	parse	execution	legal num. use	target	mean reward
Programmatic RLOO	0.996	0.996	0.996	0.930	0.613	0.791

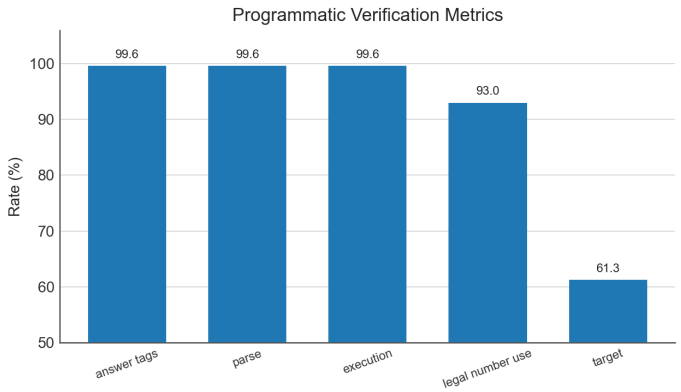

Figure 2: Programmatic verifier component metrics on Countdown test rollouts. The model nearly always emits well-formed, parseable, executable programs; the remaining errors are concentrated in legal number use and exact target success.

The diagnostics in Table 3 make the failure structure explicit: the model almost always emits a well-formed, parseable, executable program (≈ 0.996 on all three syntactic components), uses the allowed numbers legally 93% of the time, and hits the exact target on 61% of samples. In other words, once the programmatic format is learned, essentially *no* reward is lost to malformed output—the remaining errors are semantic (an illegal reuse of a number, or a legal program that simply computes the wrong value). This is the interpretability benefit RQ3 asks about: the decomposed reward attributes each lost point to a specific stage of the computation rather than collapsing every failure into a single zero. The programmatic run's training dynamics, and in particular the per-component reward trajectories, are shown in the bottom row of Figure 3.

Caveat on variance. All RL numbers are from a single seed; the pass@1 improvements are large and the training dynamics are stable, but variance across seeds is not yet quantified and should be reported alongside the extension under matched seeds.

5.2 Qualitative Analysis

SFT rollout. On the prompt with numbers $[44, 19, 35]$ and target 98, the SFT model produces a clean, direct solution: it works from the larger numbers, finds $44 + 19 = 63$ then $63 + 35 = 98$, notes an equivalent ordering, and returns `<answer> (35 + 19) + 44 </answer>`. This is representative of SFT successes—short, correct, and with little exploratory search.

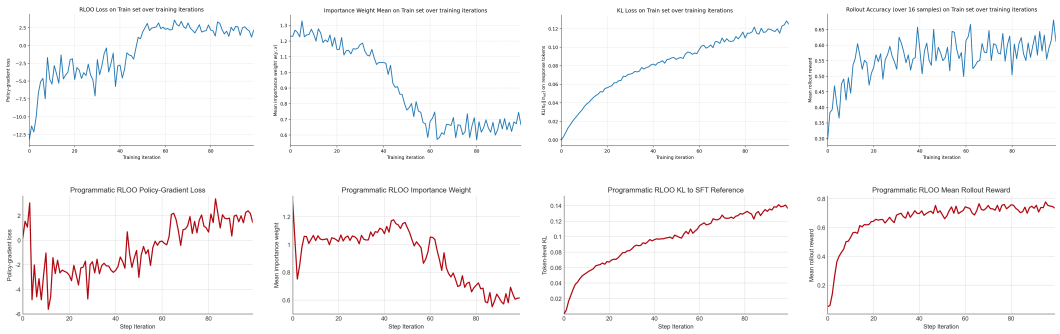

Figure 3: Training dynamics on Countdown (left to right): policy-gradient loss, mean importance weight, token-level KL to the SFT reference, and mean rollout reward over 16 samples. **Top row—standard RLOO:** loss $-13.22 \rightarrow +2.20$, importance weight $1.23 \rightarrow 0.67$ (within $[0.58, 1.33]$), KL $0 \rightarrow 0.125$, reward $0.296 \rightarrow 0.611$. **Bottom row—programmatic RLOO:** the same four diagnostics for the programmatic run; a companion panel breaks the decomposed reward into its components (format, parse, execution, legal number use, target) so each check's training-time trajectory can be read off.

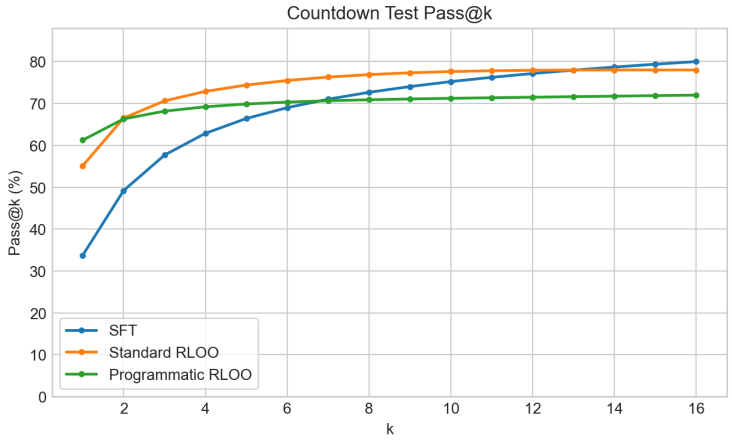

Figure 4: Pass@k on the 50-problem Countdown test set: SFT vs. standard RLOO vs. programmatic RLOO. Programmatic reward shaping gives the best pass@1, but its curve crosses below standard RLOO by pass@2 and remains lower thereafter. This indicates that the extension improves the most likely response while reducing the diversity of successful alternatives that drive larger pass@k.

RLOO rollout. The RLOO model exhibits noticeably more *search and backtracking*. On the prompt with numbers $[74, 5, 20, 88]$ and target 50, it tries several dead ends ($88-74=14,\ 14+20=34, \ldots$; division attempts that yield non-integers; products that overshoot) before discovering and verifying $(88 - 74) \times 5 - 20 = 50$, which it returns. This behavior is consistent with the training signal: rewarding only correct final answers, while penalizing drift via the KL term, pushes the policy toward trajectories that explore the operation space and self-correct—the qualitative hallmark of RL on Countdown reported in prior work Gandhi et al. (2024); Pan et al. (2025). It also illustrates the failure mode that motivates the extension: the free-form trace contains several incorrect intermediate computations that a final-answer reward cannot penalize individually.

Programmatic rollout and failure modes. The programmatic model reliably emits a parseable program. On the prompt with numbers $[44, 19, 35]$ and target 98 it returns the legal, target-hitting program `a = (44 + 19) + 35`, earning full reward. Notably, although the prompt requests one operation per line ending in an `answer` variable, the RLOO-trained policy converged to compact single-line programs of the form `a = <expression>`; the verifier accepts these because it scores the value of the final assignment. The decomposed reward makes the two dominant failure modes explicit. (i) *Illegal number use*: on $[95, 11, 56]$, target 28, the model emits `a = (56 - 11) + 95 -`

9, which parses and executes but introduces a number (9) absent from the inputs and drops one that is present; it earns only $0.05+0.10+0.15 = 0.30$ and zero for number-use and target. (ii) *Wrong target with legal numbers*: on $[83, 78, 1, 39]$, target 82, the program `a = (83 - 78) + 39 - 1` uses every number exactly once but evaluates to $43 \neq 82$, earning $0.05+0.10+0.15+0.20 = 0.50$ and zero for target. In both cases a sparse free-form reward would have returned an undifferentiated near-zero; the component reward instead pinpoints exactly which stage—number usage or final arithmetic—broke down, which is the interpretability benefit posed by RQ3. These two modes account for essentially all residual error: with parse and execution at ≈ 0.996 (Table 3), almost no reward is lost to malformed output.

6 Discussion

Limitations. Three limitations apply. First, the RL results are from a single seed; the pass@1 improvements are large and the dynamics are stable, but variance across seeds is not yet quantified. Second, the pass@k crossover shows that RL buys single-sample accuracy with a reduction in output diversity (entropy collapse), so a model that is "better" at pass@1 is not strictly better everywhere; the entropy coefficient is a lever here that we did not tune extensively, and the programmatic reward—because it also rewards parseability, execution, and number-use—pushes the operating point even further toward low-diversity, high-pass@1 behavior. Third, the off-policy gap between the vLLM sampler and the Hugging Face policy is real—the importance weight drifts to 0.67 by the end of training—and although the clip never binds in this run, larger learning rates or longer iterations would make the staleness more consequential. The 50-problem test set is also small, which inflates the variance of the pass@k estimates.

Broader impacts. Training against checkable, executable intermediate steps is a mild positive for interpretability and safety: a decomposed, execution-verified reward makes it explicit *where* a model's reasoning fails, rather than rewarding only a final answer that may be right for the wrong reasons. This is most useful in single-shot, verifier-facing deployments—for example tutoring, code assistance, or planning systems that must return one answer under latency and cost constraints—where pass@1 is more relevant than best-of-many sampling. In such settings, a downstream system can parse the program, run the verifier, reject malformed outputs, and inspect whether a failure came from syntax, execution, number use, or final arithmetic. The corresponding risk is the usual one for reward shaping: over-optimizing an easy-to-verify proxy (here, program validity) can crowd out the harder semantic objective, as the diversity collapse at large k already hints. If inference-time sampling and reranking are available, standard RLOO may be preferable because it preserves more diverse solution attempts and achieves better pass@k at larger k in our experiments.

7 Conclusion

A carefully instrumented on-policy RLOO baseline reliably improves Countdown solve rates over SFT—raising test pass@1 from ≈ 0.33 to ≈ 0.55 and more than doubling the training rollout reward—with stable, interpretable training dynamics (bounded KL, non-binding importance weights, controlled entropy decay). The key qualitative shift is that RL induces explicit search and self-correction, but it does so by trading away the output diversity that powers pass@k at large k, and its free-form traces still contain unpenalized intermediate errors. The restricted Python-style execution reward addresses the second of these problems directly: it lifts pass@1 further (to ≈ 0.61), drives program validity to near-perfect, and makes the residual failures semantically legible—but it sharpens the diversity trade-off rather than resolving it. The take-home message is that decomposed, execution-based feedback is an effective way to buy single-sample accuracy and interpretability on Countdown, and the most promising future direction is to pair it with a diversity-preserving objective (e.g. an explicit entropy or pass@k target, partial-credit shaping, or a curriculum over program length) so that the pass@1 gains no longer come at the cost of pass@k.

8 Team Contributions

- **Henry Zhou:** Implemented the programmatic reasoning extension, including the restricted Python-style verifier and the reward-shaping variant, ran evaluation, and compared the standard verifier reward against the programmatic reward variant.

- **Oleh Ivankiv:** Implemented the SFT baseline and the standard RLOO baseline, ran baseline evaluations, and compared the standard verifier reward against the programmatic reward variant.

References

Arash Ahmadian, Chris Cremer, Matthias Gallé, Marzieh Fadaee, Julia Kreutzer, Olivier Pietquin, Ahmet Üstün, and Sara Hooker. 2024. Back to Basics: Revisiting REINFORCE-Style Optimization for Learning from Human Feedback in LLMs. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. arXiv:2402.14740.

Mohammad Gheshlaghi Azar, Mark Rowland, Bilal Piot, Daniel Guo, Daniele Calandriello, Michal Valko, and Rémi Munos. 2024. A General Theoretical Paradigm to Understand Learning from Human Preferences. *Proceedings of the 27th International Conference on Artificial Intelligence and Statistics (AISTATS)* (2024). arXiv:2310.12036.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374* (2021).

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2023. Program of Thoughts Prompting: Disentangling Computation from Reasoning for Numerical Reasoning Tasks. *Transactions on Machine Learning Research* (2023).

DeepSeek-AI. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. *arXiv preprint arXiv:2501.12948* (2025).

Kanishk Gandhi, Denise Lee, Gabriel Grand, Muxin Liu, Winson Cheng, Archit Sharma, and Noah D. Goodman. 2024. Stream of Search (SoS): Learning to Search in Language. *arXiv preprint arXiv:2404.03683* (2024).

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. PAL: Program-Aided Language Models. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. 2024. RLEF: Grounding Code LLMs in Execution Feedback with Reinforcement Learning. *arXiv preprint arXiv:2410.02089* (2024).

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training Language Models to Follow Instructions with Human Feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Jiayi Pan, Junjie Zhang, Xingyao Wang, Lifan Yuan, Hao Peng, and Alane Suhr. 2025. TinyZero. `https://github.com/Jiayi-Pan/TinyZero`.

Qwen Team. 2024. Qwen2.5 Technical Report. *arXiv preprint arXiv:2412.15115* (2024).

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. 2023. Direct Preference Optimization: Your Language Model is Secretly a Reward Model. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. In *Advances in Neural Information Processing Systems (NeurIPS)*.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347* (2017).

Ronald J. Williams. 1992. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Machine Learning* 8 (1992), 229–256.