CoVER: Contrastive Self-Verification for Label-Free Reinforcement Learning

Anonymous authors
Paper under double-blind review

Abstract

A model that improves itself on unlabeled inputs would reduce its reliance on external supervision. Test-time reinforcement learning takes a step toward this goal by deriving rewards from the model's own predictions. However, agreement-based rewards only reflect the model's answer distribution: they can sharpen an existing belief, but cannot tell whether that belief is correct. This creates a fundamental form of confirmation bias. A programmatic checker provides an external correctness signal without requiring labeled training examples, but it is often sparse, task-specific, or unavailable until a useful partial solution is produced. We propose **CoVER (COntrastive VErification Reward)**, a hybrid label-free reward that combines programmatic feedback with relative self-verification. CoVER first uses contrastive verification to distinguish promising candidate solutions and maintain a useful distribution of trajectories, then incorporates programmatic feedback to provide grounded credit for valid partial progress. This combination allows the model to exploit executable evidence when available while retaining a model-derived signal when programmatic feedback is insufficient. Across diverse reasoning tasks, CoVER consistently outperforms existing label-free rewards.

1 Introduction

Improving the reasoning ability of large language models (LLMs) remains a central challenge. Recent progress has been driven in part by reinforcement learning with verifiable rewards (RLVR) (Guo et al., 2025; Shao et al., 2024; Lightman et al., 2024), which uses external signals such as ground-truth answers, unit tests, or proof assistants to guide optimization. However, such supervision is unavailable for many newly encountered inputs, particularly at test time. Test-time reinforcement learning (TTRL) addresses this setting by deriving rewards directly from the model's own predictions (Zuo et al., 2025). Given multiple rollouts of an unlabeled input, TTRL uses the plurality answer as a pseudo-label and rewards solutions that agree with it. Related approaches similarly exploit output-distribution statistics, for example by encouraging confidence (Zhao et al., 2026), minimizing predictive entropy (Agarwal et al., 2025; Prabhudesai et al., 2025), or leveraging self-consistency (Wang et al., 2023). These methods enable adaptation without external labels.

However, agreement and confidence measure what the model believes, rather than whether that belief is correct. This limitation leads to a form of confirmation bias. Because the reward depends only on the model's own answer distribution, optimization favors the model's existing modes: an answer becomes more likely because it was already likely, regardless of whether it is correct. In fact, under such rewards, any policy that deterministically outputs a single answer per input, even a wrong one, is a global optimum. Thus, frequency-based self-training can sharpen an existing belief but cannot provide information that distinguishes a confident mistake from a confident correct answer. The central question is therefore: *how can a model obtain information about the correctness of its answers without relying on external labels or merely reinforcing how frequently it produces them?*

A *programmatic checker* partially answers this question. It evaluates a candidate against the constraints stated in the problem itself, so it can score a candidate without labels or a reference solution. In addition, it can provide trajectory-level credit for solution states that are valid and make measurable progress toward a solution, even when the final answer is incorrect. Yet such feedback is often sparse, task-specific, or unavailable until the policy produces a partial solution.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Where programmatic feedback is insufficient, self-verification provides a general, model-derived source of information by asking the model not only to produce an answer, but also to evaluate candidate answers. Unlike voting, which only measures how often an answer is generated, verification asks a different conditional of the same model: given a candidate solution, how likely is it to judge that solution as correct? This additional judgment can provide information that is not captured by the model's answer frequency. However, we find that naive self-verification fails to extract this signal: when candidates are evaluated independently with a question such as "is answer a correct?", the resulting scores are nearly constant across candidates. The problem is therefore not simply whether the model can verify, but how verification is elicited. Isolated judgments require an absolute, uncalibrated decision, a setting in which language models tend to produce poorly discriminative evaluations (Huang et al., 2024; Zheng et al., 2023).

We address this problem with *contrastive verification*, which turns self-verification into a relative judgment. Rather than asking the model to independently score each candidate, we present all sampled candidates together and require the verifier to distribute a fixed amount of probability mass across them. The model must therefore compare candidates against one another, making discrimination structural rather than dependent on calibrated absolute scores. This relative verification extracts a signal beyond answer frequency and allows the model to improve upon the majority vote.

Table 1: Sources of label-free information.

	Self-verification	**Programmatic checker**
Label-free	✓	✓
General	✓	×
Task constraints	×	✓
Compare candidates	✓	Depends
Partial credit	Limited	✓

We build **CoVER (COntrastive VErification Reward)** on this idea (Figure 1). Each rollout is rewarded by the number of rollouts in its group that reach an equivalent answer, weighted by the verifier's relative preference for that answer. After a handover step, a programmatic checker grades each candidate against the problem's constraints, crediting partial progress. This grounding is needed because self-verification, even in relative form, can still reinforce a wrong consensus when the model's own judgments are systematically mistaken. Conversely, when the two rewards are mixed and every sampled solution gets the same checker score, the contrastive term still separates them. Table 1 contrasts the two sources.

Across arithmetic, combinatorial puzzles, code generation, and mathematical reasoning benchmark, CoVER consistently outperforms existing label-free rewards. Moreover, when the majority answer is wrong but a correct answer appears among the minority, CoVER selects that correct answer in 69% of such cases, whereas consensus never does and absolute self-verification does so in only 14%.

2 PRELIMINARIES

Reinforcement learning from consensus. Let π_θ be a policy over reasoning traces y for a prompt x, and let $\mathrm{ans}(y)$ extract the final answer. In RLVR the reward is $r(y) = \mathbf{1}[\mathrm{ans}(y) = y^\star]$ for a ground-truth $y^\star$ (Guo et al., 2025). Label-free test-time RL replaces $y^\star$ with a statistic of the policy's own samples. Given N rollouts $y_1, \ldots, y_N \sim \pi_\theta(\cdot \mid x)$ with answers $a_i = \mathrm{ans}(y_i)$, write $\mathcal{C}(x)$ for the set of distinct answers and $n(a) = |\{i : a_i = a\}|$ for the vote count. TTRL sets the pseudo-label to the plurality answer and rewards agreement with it (Zuo et al., 2025):

$$\hat{y}_{\mathrm{maj}}(x) \;=\; \arg\max_{a \in \mathcal{C}(x)} n(a), \qquad r_i \;=\; \mathbf{1}[a_i = \hat{y}_{\mathrm{maj}}(x)]\,. \tag{1}$$

The policy is then updated with GRPO (Shao et al., 2024), which normalizes r_i within the rollout group and applies a clipped policy-gradient step (Schulman et al., 2017). *Test-time training* (TTT) means the prompt set is the evaluation set itself: labels exist only for reporting, never for training (Sun et al., 2020; Zuo et al., 2025).

Self-verification. A *self-verifier* is the same network π_θ queried on a different input: a prompt that presents x together with one or more candidate answers and asks which are correct. We write $s(a) \in [0, 1]$ for a Monte-Carlo estimate of the verifier's confidence in candidate a, formed from K samples of the verification query.

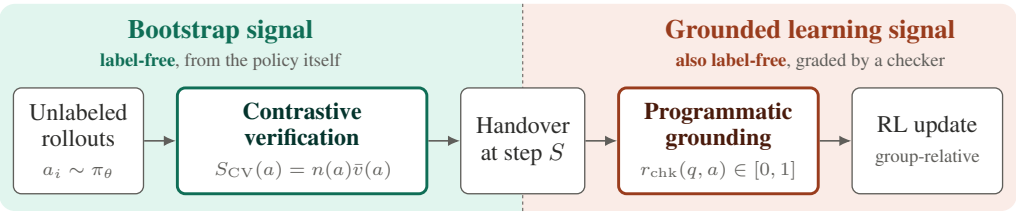

Figure 1: **Overview of our method.** Contrastive verification turns unlabeled rollouts into a *bootstrap signal*; at step S it hands over to programmatic grounding, which supplies the *grounded learning signal* for the RL update. For each unlabeled input q, the policy samples G rollouts $a_i \sim \pi_\theta(\cdot \mid q)$, whose answers are canonicalized into solution-equivalence classes. Consensus mass $n(a)$ and the relative self-verification score $\bar{v}(a)$, obtained by scoring the distinct classes jointly against one another, multiply into the contrastive verification reward $S_{\mathrm{CV}}(a)$. From step S on, a programmatic checker $r_{\mathrm{chk}}(q, a) \in [0, 1]$ supplies graded feedback on partial progress and is combined with S_{CV} in reward space or in advantage space, or replaces it. In both phases, the signal enters the update of π_θ as group-relative advantages $A(a_i)$.

3 Method

Contrastive Verification Reward (CoVER) is a novel label-free reward for test-time reinforcement learning that combines bootstrap and grounded learning signal. It has two reward components. Contrastive verification (Section 3.1) rewards answers that are supported by sibling rollouts and preferred by the verifier over the other candidates. The programmatic checker (Section 3.2) scores a candidate against constraints stated in the problem. Section 3.3 describes how the two are combined.

3.1 Contrastive Verification

For an input q, we sample a group of G rollouts and partition their answers into solution-equivalence classes. CoVER assigns each candidate answer a the contrastive verification reward

$$S_{\mathrm{CV}}(a) = n(a)\bar{v}(a), \tag{2}$$

where $n(a)$ is the number of rollouts in the group that are equivalent to a, $\bar{v}(a)$ is the verifier's relative preference for a's equivalence class. The consensus term discourages a rollout from exploiting the verifier in isolation, while the verification term provides a signal beyond answer frequency and can distinguish a correct minority answer from an incorrect majority.

Solution-equivalence classes. We map each answer to a canonical form using a task-specific normalizer. For Countdown, an expression is parsed into a syntax tree and canonicalized by flattening and sorting operands of commutative operators; thus, $(14 - 2) \times 9$ and $9 \times (14 - 2)$ belong to the same class. For Rubik's Cube, a class is defined by the permutation induced by a move sequence when applied to a solved cube; for example, `R R R` and `R'` collapse to one class. Let $n(a)$ denote the size of the class containing a. Unparseable answers are assigned unique singleton keys and therefore cannot accumulate consensus mass.

Relative self-verification. For each distinct class $a \in \mathcal{C}(q)$ the policy produces a raw verification score $s(a)$. The classes then compete for one unit of verifier mass,

$$\bar{v}(a) \;=\; \frac{\exp\big(s(a)/T\big)}{\exp\big(s_0/T\big) + \sum_{a' \in \mathcal{C}(q)} \exp\big(s(a')/T\big)} \tag{3}$$

Eq. (3) is a softmax over the candidate classes plus a null candidate with score s_0. T is the temperature.

3.2 Programmatic Checker

A *programmatic checker* is a reward function that evaluates a candidate answer using executable or otherwise hard-coded constraints derived from the problem. Unlike a conventional supervised reward, it does not compare the candidate against a stored reference answer or any hidden label. A

programmatic checker instead measures how well the candidate meets the requirements stated in the problem itself, providing an external correctness signal while remaining label-free.

Definition 1 (Programmatic checker)**.** *A reward $r_{\mathrm{chk}}(q, a) \in [0, 1]$ is a* programmatic checker *if it is a function only of the prompt string q and candidate answer a, without access to a stored reference solution, stored latent state, or any dataset field absent from q.*

Three properties follow from this definition. First, the checker is *label-free*: it can be computed on every input the policy trains on, including unlabeled test inputs and problems generated by the trainer itself. Even on generated problems, it must not read the generator's stored state; the Rubik's Cube checker, for example, recovers the scrambled cube by parsing the prompt. Second, it is *grounded*: because its value ignores how often a candidate is sampled and how the model judges it, a wrong answer earns no credit for being frequent or for convincing the model, as it can under agreement-based rewards and self-verification. Third, it can be *graded*: whereas the RLVR and TTRL rewards are binary, r_{chk} can credit valid partial progress, such as the fraction of equations in an arithmetic solution that evaluate correctly. A group in which no rollout solves the problem can therefore still produce nonzero group-relative advantages. The programmatic checker for each task is detailed in the Appendix C.

Algorithm 1 CoVER training

Require: Policy π_θ, unlabeled inputs $\mathcal{D}$, group size G, handover step S, checker weight c

for training step t **do**
  Sample $q \sim \mathcal{D}$
  Sample G rollouts $\{a_i\}_{i=1}^{G} \sim \pi_\theta(\cdot \mid q)$
  Canonicalize $\{a_i\}_{i=1}^{G}$ into equivalence classes $\mathcal{C}(q)$
  Compute class sizes $n(a_i)$ and relative verification scores $\bar{v}(a_i)$ using Eq. 3
  $r_{\mathrm{base}}(a_i) \leftarrow n(a_i)\bar{v}(a_i)$ for all i
  if $t < S$ **then**
    Compute group-relative advantages $A(a_i)$ from $r_{\mathrm{base}}(a_i)$
  else
    Compute $r_{\mathrm{chk}}(q, a_i)$ for all i
    if reward-space mixing **then**
      $r(a_i) \leftarrow r_{\mathrm{base}}(a_i) + c\, r_{\mathrm{chk}}(q, a_i)$
      Compute $A(a_i)$ from $r(a_i)$
    else if advantage-space mixing **then**
      Compute $A_{\mathrm{base}}(a_i)$ from $r_{\mathrm{base}}(a_i)$
      $A(a_i) \leftarrow A_{\mathrm{base}}(a_i) + c\big(r_{\mathrm{chk}}(q, a_i) - \overline{r_{\mathrm{chk}}}\big)$
    else ▷ replacement
      Compute $A(a_i)$ from $r_{\mathrm{chk}}(q, a_i)$
    end if
  end if
  Update θ using the group-relative advantages $\{A(a_i)\}_{i=1}^{G}$
end for

3.3 Combining Contrastive and Programmatic Rewards

We train with contrastive verification alone for S steps before introducing the programmatic checker, allowing the policy to first establish a non-degenerate candidate distribution from which meaningful comparisons can be extracted. Let $r_{\mathrm{CV}}(q, a) = S_{\mathrm{CV}}(a)$ denote the contrastive reward and $r_{\mathrm{chk}}(q, a) \in [0, 1]$ the checker reward, Norm_G denote the group-relative advantage normalization of the underlying optimizer and $c \geq 0$ the checker weight. After the handover, we combine them in one of three modes (arguments (q, a_i) omitted):

$$\mathrm{Norm}_G(r_{\mathrm{CV}} + c\, r_{\mathrm{chk}}) \quad (4a) \qquad \mathrm{Norm}_G(r_{\mathrm{CV}}) + c\ \mathrm{Norm}_G(r_{\mathrm{chk}}) \quad (4b) \qquad \mathrm{Norm}_G(r_{\mathrm{chk}}) \quad (4c)$$

In *reward-space mixing* (4a), the checker reshapes candidate rankings before normalization; c therefore absorbs the differing scales of the two rewards, since r_{CV} may grow with group size through $n(a)$ whereas r_{chk} is bounded. In *advantage-space mixing* (4b), normalizing each signal separately makes the checker act through its within-group relative ranking, removing dependence on raw reward scales and making c the relative strength of the checker signal. *Replacement* (4c) drops the contrastive reward from the post-handover update entirely, which is appropriate once the checker is reliable enough to serve as a direct learning signal.

Algorithm 1 summarizes training. At each step, we sample an unlabeled input and generate G rollouts from π_θ, canonicalize their answers into equivalence classes, and compute each class's consensus size and relative verification score. Before S, these scores determine the group-relative advantages; after S, we additionally evaluate the checker on each candidate and combine the signals according to the selected mode. The resulting advantages update π_θ using the policy-optimization algorithm.

4 EXPERIMENTS

We evaluate our proposed method on a diverse suite of challenging tasks. Detailed task configurations are discussed in the Appendix A.

4.1 BASELINES.

We evaluate CoVER against the several strong representative baseline. **Base model:** the pretrained model without additional test-time training, serving as our main baseline. **TTRL**: uses the majority answer among sampled rollouts as a pseudo-label and assigns a binary reward to each rollout based on whether its answer agrees with this majority (Zuo et al., 2025). **Intuitor**: reward using the model's own self-certainty from the online policy and using the group-normalized self-certainty for policy gradient update (Zhao et al., 2026). **EMPO**: the reward for each sampled response is the probability mass of its semantic cluster, estimated via clustering with dual entropy thresholding (Zhang et al., 2025b). **Co-rewarding**: derives pseudo-label rewards from cross-view agreement between the online policy and a slowly EMA-updated teacher to break the single-view entanglement (Zhang et al., 2026).

4.2 ARITHMETIC

Setup. We evaluated CoVER against baseline methods on the Arithmetic task. The task gives the model a symbolic arithmetic expression as input and asks it to generate the final numeric result as the output, with the target being the exact correct value of that expression. That includes addition, subtraction, multiplication, and multi-step expressions. We trained all models with GRPO on 960-problem training set and tested them on a disjoint 200-problem held-out slice, thereby measuring each method's ability to generalize to unseen arithmetic problems.

Result. As shown in Table 2, CoVER achieves the highest held-out accuracy of all methods (0.688). This is an absolute gain of 17.5 percentage points (34% relative) over the base model and 4.4 points over the strongest baseline, Co-rewarding (0.644). All label-free baselines also improve on the base model, by 7.4 to 13.1 points. This is because the base model already solves about half of the problems, so sharpening its own answer distribution is often beneficial. Contrastive verification shifts reward toward candidates that survive an inverse-operation check: for a candidate answer c to $a + b$, the model is asked for $c - b$ and should recover a. The programmatic checker credits the fraction of equations in a solution that evaluate correctly. CoVER also has the lowest variance across runs ($\pm$0.009, compared with $\pm$0.025-0.065 for the baselines), indicating that its improvement is consistent. Figure 2 shows the learning curve of all methods. Section 5 analyzes the contribution of each reward component.

4.3 COUNTDOWN

Setup. We evaluated CoVER against baseline methods on the Countdown arithmetic task. The Countdown task gives the model three integers sampled from 1–12 and a target integer in the range 10–60. The model must produce an arithmetic expression that uses the provided numbers, matching the given multiset, and evaluates to the target. We trained all models on the training split using the GRPO with a disjoint 200-problem validation slice for checkpoint selection, and tested them on a held-out 200-problem test slice.

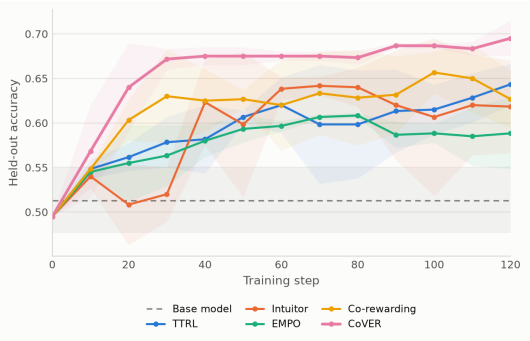

Figure 2: **Arithmetic task**. Learning curve of all methods.

Result. As shown in Table 3, CoVER raises held-out accuracy from 0.046 to 0.471, an absolute gain of 42.5 percentage points over the base model. Greedy accuracy rises from 0.039 to 0.475, and Pass@4 from 0.155 to 0.536. The

Method	Accuracy
Base model	0.513 ± 0.036
TTRL	0.629 ± 0.025
Intuitor	0.615 ± 0.065
EMPO	0.587 ± 0.028
Co-rewarding	0.644 ± 0.031
CoVER (ours)	$\mathbf{0.688 \pm 0.009}$

Table 2: Arithmetic.

Method	Accuracy	Greedy	Pass@4
Base model	0.046 ± 0.009	0.039 ± 0.006	0.155 ± 0.030
TTRL	0.053 ± 0.007	0.036 ± 0.006	0.161 ± 0.016
Intuitor	0.110 ± 0.083	0.109 ± 0.086	0.122 ± 0.090
EMPO	0.051 ± 0.009	0.038 ± 0.008	0.142 ± 0.048
Co-rewarding	0.051 ± 0.006	0.037 ± 0.008	0.148 ± 0.028
CoVER (ours)	$\mathbf{0.471 \pm 0.221}$	$\mathbf{0.475 \pm 0.224}$	$\mathbf{0.536 \pm 0.277}$

Table 3: Countdown.

baselines have almost nothing correct to reinforce. TTRL, EMPO and Co-rewarding reward agreement with the model's most frequent answers, but the base model is wrong on more than 95% of samples. The majority vote of Co-rewarding's teacher, for instance, is correct only 6.3% of the time. This is the confirmation bias discussed in the introduction. Intuitor can improve accuracy, but it does not raise Pass@4, which suggests its gain comes from answering more consistently on problems the base model could already solve rather than from solving new ones. On Countdown, checking a solution is much easier than finding one. Reaching the target requires search over combinations of numbers and operators, whereas evaluating a given expression takes a few steps of arithmetic. CoVER recomputes each candidate's value and checks whether it uses each given number exactly once, so a correct expression sampled once can outrank a wrong one sampled several times.

4.4 Rubik's Cube

Setup. We also evaluated CoVER against baseline methods on the Rubik's Cube task. The task asks the model to output a sequence of canonical Rubik's-move tokens that, when applied to a scrambled cube shown in the prompt, transforms it to the solved state. Answers are canonicalised by the permutation the move sequence induces on a solved cube. Correctness is checked by applying the candidate moves to compare resulting cube states. We trained all models with GRPO on 4000-problem training set and tested them on a disjoint 200-problem held-out slice, thereby measuring each method's ability to improve exact-solve rate.

Result. As shown in Table 6, CoVER raises held-out accuracy from 0.023 to 0.167, an absolute gain of 14.4 percentage points over the base model. Greedy accuracy doubles from 0.083 to 0.167, and Pass@4 rises from 0.088 to 0.214. The label-free baselines improve the base model slightly. CoVER credits each legal move sequence with the relative reduction in misplaced facelets it achieves, which suits a task where full solves are rare. A rollout that leaves the cube unsolved but with fewer misplaced facelets still outscores siblings that make less progress, so even groups without a single solve provide a learning signal. The gains are not limited to cubes the base model already solves within four attempts: a single CoVER sample succeeds more often (0.167) than the base model does with four attempts (0.088 Pass@4) or than any baseline does (at most 0.110). The improvement is also consistent.

4.5 LiveCodeBench

Setup. We evaluated CoVER against baseline methods on the LiveCodeBench code generation benchmark. The task asks the model to write a Python program that solves a competitive-programming problem. A sample counts as correct only if it passes every public and private test. We trained all models with GRPO on 2978-problem training set of standard-input problems. We tested on a disjoint 1055-problem set, thereby measuring each method's ability to improve coding performance.

Result. As shown in Table 4, CoVER achieves the highest overall Pass@1 (13.34%), an absolute gain of 2.51 percentage points (23% relative) over the base model. All label-free baselines also improve on the base model overall, but only by 0.14 to 1.85 points. CoVER's largest gain is on the functional split, where it raises Pass@1 from 12.44% to 16.77%, 1.07 points above EMPO (15.70%). Because all training problems use standard input, these call-based problems use a format never seen during training, so the improvement transfers beyond the training format. On the standard-input split,

Method	avg@8
Base model	10.42 ± 1.63
TTRL	21.08 ± 1.36
Intuitor	17.98 ± 0.49
EMPO	21.51 ± 1.79
Co-rewarding	20.81 ± 0.21
CoVER (ours)	$\mathbf{23.52 \pm 0.56}$

Table 5: AIME.

Method	Accuracy	Greedy	Pass@4
Base model	0.023 ± 0.002	0.083 ± 0.002	0.088 ± 0.007
TTRL	0.027 ± 0.013	0.063 ± 0.026	0.090 ± 0.025
Intuitor	0.027 ± 0.007	0.089 ± 0.010	0.098 ± 0.023
EMPO	0.030 ± 0.008	0.054 ± 0.023	0.104 ± 0.021
Co-rewarding	0.032 ± 0.010	0.070 ± 0.023	0.110 ± 0.021
CoVER (ours)	$\mathbf{0.167 \pm 0.011}$	$\mathbf{0.167 \pm 0.014}$	$\mathbf{0.214 \pm 0.014}$

Table 6: Rubik's Cube.

which is harder for the base model (9.66% vs. 12.44%), CoVER gains 1.19 points and is on par with the strongest baseline on this split, Co-rewarding (10.87%). TTRL and Intuitor do not improve on the base model there at all.

4.6 AIME

Setup. We evaluated CoVER against baseline methods on AIME mathematical reasoning benchmark. The task gives the model a competition math problem and asks it to reason step by step to a final integer answer between 0 and 999, which counts as correct only if it exactly matches the official answer. We train and evaluate on the 30 problems of AIME 2024. Ground-truth answers are used only for evaluation. We train all methods with GRPO. Additional experimental details are provided in Appendix.

Method	functional	stdin	all
Base model	12.44%	9.66%	10.83%
TTRL	15.01%	9.34%	11.72%
Intuitor	12.88%	9.57%	10.97%
EMPO	15.70%	10.50%	12.68%
Co-rewarding	13.29%	10.87%	11.89%
CoVER	**16.77%**	10.85%	**13.34%**

Table 4: LiveCodeBench.

Result. As shown in Table 5, CoVER achieves the highest avg@8 of all methods (23.52). This is an absolute gain of 13.10 points over the base model, more than doubling its accuracy, and 2.01 points over the strongest baseline, EMPO (21.51). All label-free baselines also improve on the base model, by 7.56 to 11.09 points. CoVER adds two signals that frequency does not provide. First, the policy combines verification score $\bar{v}(a)$ with its consensus mass $n(a)$, so a correct minority answer can outscore a wrong plurality. Second, a programmatic checker grades the derivation rather than the answer. An AIME answer cannot in general be verified from the problem statement alone. Each rollout restates its solution as a numbered proof trace and is credited with the fraction of claims that are executable and entailed by their cited premises. Because rollouts that reach the same answer share one contrastive reward, the checker is what separates them, favoring those whose derivations check out. CoVER's spread across runs (± 0.56) is also less than half that of the two strongest baselines, EMPO (± 1.79) and TTRL (± 1.36).

5 Ablation

5.1 Programmatic feedback is an effective foundation for label-free RL.

Setup. To isolate the contribution of each component of our method, we keep the training protocol fixed and vary only the reward signal. All runs use the same GRPO setup, model, and evaluation protocol: Qwen2.5-0.5B-Instruct trained on the arithmetic task with 120 optimization steps, batch size 8, group size 8, and held-out evaluation on 200 problems with greedy decoding. We compare four variants: consensus-only, verifier-only, pro-

Method	Accuracy
Base model	0.513 ± 0.036
VERIFY only, score $= \bar{v}(a)$	0.563 ± 0.018
CONSENSUS only, score $= n(a)$	0.586 ± 0.059
PROGRAMMATIC CHECKER only	0.675 ± 0.015
CoVER	$\mathbf{0.688 \pm 0.009}$

Table 7: Accuracy on the final evaluation.

grammatic checker-only, and the full CoVER method. The base model without RL is included as a reference.

Result. Table 7 summarizes the performance for each ablation component. As we can see, programmatic feedback provides the strongest individual learning signal. On its own, it raises held-out accuracy from 0.513 to 0.675, a gain of 16.2 percentage points. Since this score depends only on the prompt and the candidate, a wrong answer earns no credit for being frequent. The checker also makes training more stable across runs. These indicate that programmatic feedback is the foundation of the learning signal. Combining it with contrastive verification, raises accuracy by a further 1.3 points to 0.688 and yields the smallest spread of all variants (± 0.009).

5.2 Contrastive Verification Can Recover Minority Solutions

Setup. The key design of CoVER is to improve the limitation of majority voting. It is natural to ask: *when a group's majority answer is wrong and a correct answer is present in the minority, can CoVER find the correct minority?* We measure this directly on frozen rollouts on Countdown task, so the comparison isolates the reward's selection behaviour from the optimisation dynamics it would induce. We sample from the base Qwen2.5-1.5B-Instruct policy. **Consensus n(a)** score every class by how many of the G rollouts landed in it. **Absolute self-verification** asks the model instead of counting: a per-candidate Yes/No query, each class judged alone. **CoVER** consensus multiplied by a verifier term, where $\bar{v}$ is the competition-normalised re-derivation score, so classes compete for one unit of mass. More details are provided in the Appendix D.

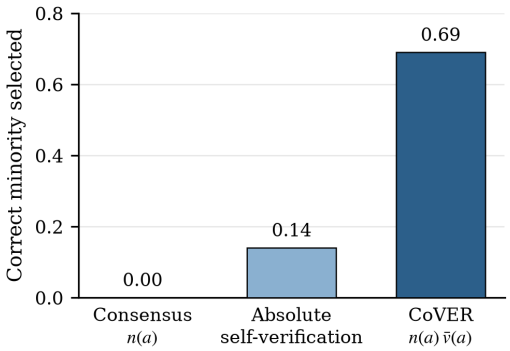

Figure 3: **Correct minority selected.**

Result. As shown in Figure 3, consensus favors an incorrect solution. Absolute self-verification also fails to identify the correct solution. It actually assigns a higher score to an invalid candidate, as show in Table 10. In contrast, CoVER performs relative verification across the candidate set and assigns substantially different probability mass to the solution classes. Thus, it can overturn a wrong majority and recover a correct solution that would otherwise be suppressed by both consensus and absolute verification.

5.3 Effect of Reward Handover Strategy

Setup. CoVER introduces its two reward sources one after the other: contrastive verification first builds a spread of candidates, then the programmatic checker adds grounded credit. To test whether this order matters, we keep the arithmetic protocol of Section 5.1 fixed. Only the schedule of the reward signal changes. We compare four schedules: (1) Contrastive Verification only: $r = S_{\mathrm{CV}}(a)$ for the whole run. (2) Programmatic Checker only: $r = r_{\mathrm{chk}}(q, a)$ for the whole run. (3) Programmatic Checker $\rightarrow$ Contrastive Verification (reversed order): the checker is the only reward for the first phase. In the second phase, the checker is turned off and training continues with S_{CV} alone. (4) Contrastive Verification $\rightarrow$ Programmatic Checker (CoVER): S_{CV} is the only reward for the first phase. After that, the checker is added through reward-space mixing Eq. (4a).

Method	Accuracy
CV only	0.619 ± 0.070
PC only	0.675 ± 0.015
PC $\rightarrow$ CV	0.686 ± 0.015
CV $\rightarrow$ PC	$\mathbf{0.688 \pm 0.009}$

Table 8: Accuracy of Reward Handover Strategy. CV: Contrastive Verification; PC: Programmatic Checker.

Result. As shown in Table 8, contrastive verification alone raises accuracy from 0.513 to 0.619, but it is the least stable schedule. By contrast, every schedule that uses the checker is more accurate and more stable. Both two-phase schedules exceed the checker alone (0.686 and 0.688 vs. 0.675), so combining the two signals helps. The reversed schedule matches checker-only training in stability and slightly exceeds it in accuracy. This suggests that the instability of contrastive verification alone comes from the absence of grounding rather than from the verification signal itself. Finally, CoVER attains the highest accuracy and the smallest variance of all schedules.

6 Related work

Label-free and test-time reinforcement learning. TTRL replaces the checker with the policy's own majority vote (Zuo et al., 2025), extending test-time training (Sun et al., 2020) and self-consistency (Wang et al., 2023) into the RL loop. Neighboring proposals reward the policy's confidence (Zhao et al., 2026), minimize its predictive entropy (Agarwal et al., 2025; Prabhudesai et al., 2025), or remove the data entirely via self-play (Zhao et al., 2025). We make the reward depend on a *second* conditional of the same network.

Self-verification and model-as-judge. Scoring outputs with a model underpins RLAIF (Lee et al., 2024), constitutional training (Bai et al., 2022), self-rewarding language models (Yuan et al., 2024), generative verifiers (Zhang et al., 2025a) and process reward models (Lightman et al., 2024). The known limits are directly relevant: models correct their own reasoning poorly without external feedback (Huang et al., 2024), their absolute self-assessments are badly calibrated relative to elicited or comparative ones (Tian et al., 2023), and judges show strong position and format effects (Zheng et al., 2023; Wang et al., 2024).

Learning from unlabeled data. Consensus RL is pseudo-labeling (Gkountouras et al., 2026) with a policy-gradient outer loop, and it inherits pseudo-labeling's confirmation bias (Arazo et al., 2020). Classical theory says the escape needs two sufficiently independent views and quantifies the learnable signal by their disagreement (Blum & Mitchell, 1998; Dasgupta et al., 2001); recent analysis instead grounds self-training in an expansion property of the unlabeled distribution (Wei et al., 2021). Our two channels, solve and judge, are two views inside one set of weights. Entropy minimization (Grandvalet & Bengio, 2004) encourages confident predictions on unlabeled data by minimizing predictive entropy, but can be harmful when the unlabeled data provide little or misleading information.

7 Limitations and Discussion

While CoVER performs well on tasks whose outputs can be canonicalized and checked by execution, its applicability to open-ended outputs, such as informal proofs or long-form writing remains unexplored. In addition, CoVER depends on task structure: the raw verification score, the checker, and the schedule (S, c, mixing mode) are all set per task. Finally, solver and verifier share weights, so they are not the independent views that co-training theory requires.

Several promising future directions follow from our results. The per-task engineering for checker could be reduced by letting the policy write executable checks from the problem statement. Where a checker exists, its signal could also supervise the self-verifier rather than only the solver, which may yield a verifier that transfers to related tasks without one. Finally, testing CoVER with larger models in interactive environment would show whether its gains hold at scale.

8 Conclusion

We introduced CoVER, a label-free reward that lets a policy improve on unlabeled inputs. It frames self-verification as a relative judgment between candidates to give it a signal that can overturn a wrong majority, and it then grounds learning with programmatic credit for valid partial progress. Across arithmetic, combinatorial puzzles, code generation, and mathematical reasoning, CoVER consistently outperforms existing label-free rewards, suggesting that, in the absence of labels, verification matters more than generation.

AI USE STATEMENT

In this work, we used generative AI tools to improve the writing and presentation of this paper, a task for which the ICLR 2027 AI Policy for Authors recommends disclosure. We did not use generative AI tools for any of the tasks for which the policy requires disclosure. Specifically, no generative AI tool was used to create synthetic datasets, help develop theoretical models or conceptual frameworks, formulate or help prove mathematical claims, propose or refine hypotheses, design or give feedback on our methodology or experiments, implement our methods, translate text, clean or reformat datasets, assist with qualitative or thematic data analysis, or interpret our results.

We have reviewed all AI-assisted work. All text was written by the authors; the tools suggested improvements to grammar, wording, and clarity but did not contribute any ideas, technical content, claims, results, or references. Every suggested edit was checked by the authors and kept only if it preserved the intended meaning and technical accuracy of the original text. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work involves no human subjects, crowdsourced annotation, or personal data. We use open-weight Qwen2.5 models and public benchmarks (AIME 2024, LiveCodeBench) in accordance with their licenses and terms of use. All other tasks (Arithmetic, Countdown, and Rubik's Cube) are synthetic. Model-generated code was executed only in a sandbox with per-test time limits. The main risk we foresee is inherent to label-free self-improvement. It reduces human oversight of the learning signal and, where no reliable checker exists, can reinforce a model's systematic errors or biases. CoVER's programmatic grounding targets this failure mode, but models trained this way should still be validated independently before high-stakes use. Our experiments are limited to verifiable reasoning tasks, so we foresee no misuse risks beyond those of general improvements in reasoning.

REPRODUCIBILITY STATEMENT

Section 3 specifies CoVER: the contrastive verification reward (Eqs. 2–3), the three post-handover mixing modes (Eq. 4), and the training loop (Algorithm 1). Section 4 and Appendix A give the task definitions and data splits, base models, and per-task training settings, including the handover step S, checker weight c, and mixing mode. They also give the evaluation metrics and the hyperparameter search protocol used for the baselines. Appendix B defines the raw verification score for each task, Appendix C gives each task's programmatic checker, and Appendix D details the frozen-rollout analysis behind Figure 3. Figures 4–12 reproduce the prompts used for generation and verification. All base models (Qwen2.5) and benchmarks (AIME 2024, LiveCodeBench) are publicly available.

REFERENCES

Shivam Agarwal, Zimin Zhang, Lifan Yuan, Jiawei Han, and Hao Peng. The unreasonable effectiveness of entropy minimization in LLM reasoning. *arXiv preprint arXiv:2505.15134*, 2025.

Eric Arazo, Diego Ortego, Paul Albert, Noel E. O'Connor, and Kevin McGuinness. Pseudo-labeling and confirmation bias in deep semi-supervised learning. In *International Joint Conference on Neural Networks*, 2020.

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, et al. Constitutional AI: Harmlessness from AI feedback. *arXiv preprint arXiv:2212.08073*, 2022.

Avrim Blum and Tom Mitchell. Combining labeled and unlabeled data with co-training. In *Conference on Computational Learning Theory*, 1998.

Sanjoy Dasgupta, Michael L. Littman, and David McAllester. PAC generalization bounds for co-training. In *Advances in Neural Information Processing Systems*, 2001.

John Gkountouras, Josip Jukić, and Ivan Titov. Consensus as privileged context for label-free self-distillation. *arXiv preprint arXiv:2607.13643*, 2026.

Yves Grandvalet and Yoshua Bengio. Semi-supervised learning by entropy minimization. In *Advances in Neural Information Processing Systems*, 2004.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. *International Conference on Learning Representations*, 2024.

Harrison Lee, Samrat Phatale, Hassan Mansoor, Thomas Mesnard, Johan Ferret, Kellie Lu, Colton Bishop, Ethan Hall, Victor Carbune, Abhinav Rastogi, and Sushant Prakash. RLAIF vs. RLHF: Scaling reinforcement learning from human feedback with AI feedback. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, volume 235 of *Proceedings of Machine Learning Research*, pp. 25010–25039. PMLR, 2024. URL `https://proceedings.mlr.press/v235/lee24t.html`.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *International Conference on Learning Representations*, 2024.

Mihir Prabhudesai, Lili Chen, Alex Ippoliti, Katerina Fragkiadaki, Hao Liu, and Deepak Pathak. Maximizing confidence alone improves reasoning. *arXiv preprint arXiv:2505.22660*, 2025.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Yu Sun, Xiaolong Wang, Zhuang Liu, John Miller, Alexei A. Efros, and Moritz Hardt. Test-time training with self-supervision for generalization under distribution shifts. In *International Conference on Machine Learning*, 2020.

Katherine Tian, Eric Mitchell, Allan Zhou, Archit Sharma, Rafael Rafailov, Huaxiu Yao, Chelsea Finn, and Christopher D. Manning. Just ask for calibration: Strategies for eliciting calibrated confidence scores from language models fine-tuned with human feedback. In *Conference on Empirical Methods in Natural Language Processing*, 2023.

Peiyi Wang, Lei Li, Liang Chen, Zefan Cai, Dawei Zhu, Binghuai Lin, Yunbo Cao, Lingpeng Kong, Qi Liu, Tianyu Liu, and Zhifang Sui. Large language models are not fair evaluators. In *Annual Meeting of the Association for Computational Linguistics*, 2024.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *International Conference on Learning Representations*, 2023.

Colin Wei, Kendrick Shen, Yining Chen, and Tengyu Ma. Theoretical analysis of self-training with deep networks on unlabeled data. In *International Conference on Learning Representations*, 2021.

Weizhe Yuan, Richard Yuanzhe Pang, Kyunghyun Cho, Xian Li, Sainbayar Sukhbaatar, Jing Xu, and Jason Weston. Self-rewarding language models. In *International Conference on Machine Learning*, 2024.

Lunjun Zhang, Arian Hosseini, Hritik Bansal, Mehran Kazemi, Aviral Kumar, and Rishabh Agarwal. Generative verifiers: Reward modeling as next-token prediction. In *International Conference on Learning Representations*, 2025a.

Qingyang Zhang, Haitao Wu, Changqing Zhang, Peilin Zhao, and Yatao Bian. Right question is already half the answer: Fully unsupervised llm reasoning incentivization. *Advances in neural information processing systems*, 2025b.

Zizhuo Zhang, Jianing Zhu, Xinmu Ge, Zihua Zhao, Zhanke Zhou, Xuan Li, Xiao Feng, Jiangchao Yao, and Bo Han. Co-rewarding: Stable self-supervised rl for eliciting reasoning in large language models. In *International Conference on Learning Representations (ICLR)*, 2026. URL https://arxiv.org/abs/2508.00410.

Andrew Zhao, Yiran Wu, Yang Yue, Tong Wu, Quentin Xu, Yang Yue, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. Absolute zero: Reinforced self-play reasoning with zero data. In *Advances in Neural Information Processing Systems*, 2025.

Xuandong Zhao, Zhewei Kang, Aosong Feng, Sergey Levine, and Dawn Song. Learning to reason without external rewards. In *International Conference on Learning Representations*, volume 2026, pp. 2548–2581, 2026.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. In *Advances in Neural Information Processing Systems*, 2023.

Yuxin Zuo, Kaiyan Zhang, Li Sheng, Shang Qu, Ganqu Cui, Xuekai Zhu, Haozhan Li, yuchen zhang, Xinwei Long, Ermo Hua, Biqing Qi, Youbang Sun, Zhiyuan Ma, Lifan Yuan, Ning Ding, and Bowen Zhou. Ttrl: Test-time reinforcement learning. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 131459–131483. Curran Associates, Inc., 2025. doi: 10.52202/085713-4376. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/be690ea16f005c174f6c4102a5970e67-Paper-Conference.pdf.

A EXPERIMENTAL SETUP IN DETAIL

Arithmetic The base model is Qwen2.5-0.5B-Instruct. The metric as final3, defined as the mean of the last three held-out evaluation accuracies, which smooths out spikes from any single checkpoint. S=10. $c = 0.3$.

Countdown The base model is Qwen2.5-1.5B-Instruct. S=20. Advantage-space mixing. Held-out accuracy: the fraction of exactly-solved problems on 200 unseen Countdown instances, averaged over 4 samples per problem. Greedy: Exact-solve rate on one temperature-0 sample per problem. Pass@4: Fraction of problems solved by at least one of 4 samples.

Rubik's Cube The base model is Qwen2.5-1.5B-Instruct. S=10. Replacement. 100 GRPO steps. 16 prompts × 8 rollouts per step. 2×2×2 scrambled cubes. Held-out accuracy: the fraction of exactly-solved problems on 200 unseen 2x2x2 Rubik's cube scrambles (1–2 moves), averaged over 4 samples per problem. Greedy: the exact-solve rate on one temperature-0 sample per held-out cube scramble. Pass@4: Fraction of cubes solved by at least one of 4 samples.

LiveCodeBench The base model is Qwen2.5-1.5B. Pass@1: averaged over 16 samples per problem at temperature 0.2. Reward space mixing. $c = 8$. The test set has 444 functional (LeetCode, call-based) problems and 611 standard-input (AtCoder and Codeforces) problems. 8 prompts $\times$ 8 rollouts per step, rollout temperature 1.0, max 900 response tokens. Learning rate 2×10^{-6} with 10 warmup steps, no KL penalty, $\epsilon = 0.2$. $S = 0$, activating both rewards from the first step. 4 self-generated probe inputs per problem define the equivalence classes. 120 GRPO steps. Execution sandbox: 4 s per test during training, 6 s at evaluation.

AIME The base model is Qwen2.5-Math-7B. Each training step uses 10 problems. For each, 16 rollouts are sampled at temperature 1.0 to form the candidate set, and 8 of them are used for the policy update. Learning rate of 5×10^{-7} with cosine decay, a KL-loss coefficient of 10^{-3}, and a maximum response length of 3072 tokens. Rollouts are grouped into solution-equivalence classes by their normalized final answer. K=4 largest classes enter the contrastive verification prompt in Figure 9. M=8 verification rollouts per problem. $\epsilon = 0.2$. S=30. 55 GRPO steps. Reward space mixing. $c = 0.1$. Avg@8: Mean accuracy over 8 sampled responses per problem.

Hyperapameter For hyperparameter selection of the baselines, each was a staged ladder of 3 rounds, start from the specified config and change one hyperparamter per run, the configuration with the highest mean held-out accuracy across its 3 seeds, followed by a 12-seed confirmation round.

Countdown

```
Evaluate this arithmetic expression step by step:  {candidate}
Then check whether it uses each of the numbers {numbers} exactly once.
Finish with exactly these two lines:
VALUE: <the number the expression evaluates to>
USESALL: <yes or no>
```

Figure 4: Prompt used to evaluate a candidate expression.

B RAW VERIFICATION SCORE

Arithmetic. The raw verification score s(a) is an inverse-operation pass rate. For a problem a op b and a candidate answer c, the check inverts the operation and asks whether the model lands back on the first operand a:

original	inverse sub-problem asked	must produce
`a + b = c`	"What is `c - b`?"	`a`
`a - b = c`	"What is `c + b`?"	`a`
`a * b = c`	"What is `c / b`?"	`a`

That sub-problem is sampled `verify_samples` times at the training temperature (1.0), capped at 160 tokens. Each sample is a pass if the extracted answer is within 1e-6 of a.

$$s(a) = \frac{\#\{\text{samples whose answer} = a\}}{\texttt{verify_samples}} \tag{5}$$

Countdown. The verifier checks whether the expression reaches the target and uses every source number exactly once:

$$s(a) = 2 \cdot \mathbf{1}[\text{value} = \text{target}] + \mathbf{1}[\text{uses all numbers}] + \left(1 + \frac{|\text{value} - \text{target}|}{\max(1, |\text{target}|)}\right)^{-1}. \tag{6}$$

Rubik's Cube. The raw score is a logit margin read off the last position of one forward pass.

$$s_{raw}(\text{trajectory}) = \text{logit("Yes")} - \text{logit("No")} \tag{7}$$

LiveCodeBench. The raw verification score is s(a) = P("Yes"), the policy's own next-token probability of "Yes" vs "No" on the verification prompt.

AIME. The raw verification score s(a) is the number of verification votes a solution class wins. Each prompt's `n_votes` rollouts are grouped by normalized final answer (classes with counts n(a)). The top classes go into one contrastive prompt. `verify_n` verification rollouts are drawn. Each reply is parsed to a slot index or an echoed answer. We use the share over total valid votes as the probability distribution over candidates.

When calculating the contrastive verification score $S_{\mathrm{CV}}(a)$, we add $\epsilon > 0$ to the verifier probability mass $\bar{v}(a)$ to ensure that consensus remains useful even when verifier confidence is low.

Rubik's Cube

```
{problem}
Here is an attempt by another student:
--
{trajectory}
--
Is the attempt's move sequence correct -- does applying it to the
cube above leave every face a single color?  Reply Yes or No.
```

Figure 5: Prompt used to query the verifier.

Rubik's Cube generation prompt example

```
Example. A cube scrambled with 1 random face turn from solved looks
    like this:

       W  G
       W  G
 O  O  G  Y  R  R  W  B
 O  O  G  Y  R  R  W  B
       Y  B
       Y  B


Only the R turn was applied, so undoing it solves the cube: <answer>R
    '</answer>

Example. A cube scrambled with 1 random face turn from solved looks
    like this:

       W  W
       W  W
 B  B  O  O  G  G  R  R
 O  O  G  G  R  R  B  B
       Y  Y
       Y  Y


Only the U' turn was applied, so undoing it solves the cube: <answer>U
    </answer>

You see a size 2 Rubik's cube arranged as follows::

       W  W
       W  W
 G  G  R  R  B  B  O  O
 G  G  R  R  B  B  O  O
       Y  Y
       Y  Y


Please provide a solution to solve this cube using Singmaster notation
    . Do not combine any steps, for instance, do not write 'U2', and
    instead write 'U U'.

The cube was scrambled with between 1 and 2 random face turn(s) from
    the solved state, so a short solution exists.
Use only the moves L, R, U, D, F, B and their primes L' R' U' D' F' B
    ', separated by spaces. Do not write U2; write U U.
Reason briefly, then finish your reply with the move sequence inside <
    answer></answer> tags, for example <answer>R U' F</answer>.
```

Figure 6: Prompt for the 2×2 Rubik's cube task, with two one-move demonstrations followed by the query cube.

LiveCodeBench

```
### Problem
{question}

### Candidate program
```python
{code}
```

Observed behavior
On the input:
{input}
the program printed:
{output}

Question
Is this candidate program a correct solution to the problem? Answer
    Yes or No.
Answer:
```

Figure 7: Prompt used to verify a candidate program.

Arithmetic

Query construction

```
inv = {"+": "−", "−": "+", "*": "/"}[op]
return f"What is {fmt(candidate)} {inv} {b}?", a
```

Resulting prompt

```
<|im_start|>system
You are a careful calculator. Solve the arithmetic problem step by
step, then give the final answer on the last line in the form
'Answer: <number>'.<|im_end|>
<|im_start|>user
What is 271 − 130?<|im_end|>
<|im_start|>assistant
```

Figure 8: The verifier re-asks the problem with the operation inverted; the expected answer is the original first operand a.

AIME contrastive verification

```
{question}

{n} independent attempts at the problem above finished with the
following candidate answers. At most one of them is correct.

{candidates}
Do not trust the attempts. Check each candidate against the problem
statement: re-derive the key quantities, check the arithmetic and the
algebra of the excerpt, and rule out the candidates that are
inconsistent. Then decide which single candidate is correct.

End your reply with the index of the correct candidate in \boxed{}.
```

Format of each entry in `{candidates}`

```
[{i}] Candidate answer: {answer}
Excerpt from the attempt that produced it:
{excerpt}
```

Figure 9: Prompt used to select the correct answer among n candidates. Each candidate is rendered with the template in the lower panel and the results are concatenated into `{candidates}`.

AIME phase 1 generation prompt

```
<|im_start|>system
Please reason step by step, and put your final answer within \boxed
    {}.<|im_end|>
<|im_start|>user
Every morning Aya goes for a $9$-kilometer-long walk and stops at a
    coffee shop afterwards. When she walks at a constant speed of $s$
    kilometers per hour, the walk takes her 4 hours, including $t$
    minutes spent in the coffee shop. When she walks $s+2$ kilometers
    per hour, the walk takes her 2 hours and 24 minutes, including $t$
     minutes spent in the coffee shop. Suppose Aya walks at $s+\frac
    {1}{2}$ kilometers per hour. Find the number of minutes the walk
    takes her, including the $t$ minutes spent in the coffee shop.<|
    im_end|>
<|im_start|>assistant
```

Figure 10: An example AIME problem rendered with the chat template, exactly as it is fed to the model.

AIME phase 2 generation prompt

```
<|im_start|>system
Please reason step by step, and put your final answer within \boxed
    {}.<|im_end|>
<|im_start|>user
Every morning Aya goes for a $9$-kilometer-long walk and stops at a
    coffee shop afterwards. When she walks at a constant speed of $s$
    kilometers per hour, the walk takes her 4 hours, including $t$
    minutes spent in the coffee shop. When she walks $s+2$ kilometers
    per hour, the walk takes her 2 hours and 24 minutes, including $t$
     minutes spent in the coffee shop. Suppose Aya walks at $s+\frac
    {1}{2}$ kilometers per hour. Find the number of minutes the walk
    takes her, including the $t$ minutes spent in the coffee shop.

After the solution, restate the derivation as a checkable proof trace:
     a block of numbered
claims, each one a single equation (or chain of equations), each
    citing where it comes from.
Cite `problem` for an equation read directly off the problem statement
    , otherwise cite the
numbers of the earlier claims it follows from. Only use a symbol after
     a claim you cite has
given it a value. Put no prose inside the block. For example:

<claims>
1. [premises: problem] 4*s = 9
2. [premises: 1] s = 9/4
3. [premises: problem] t = 24
4. [premises: 2, 3] 2*s + t = 9/2 + 24 = 57/2
</claims>

Then, on the last line, give the final answer in \boxed{}.<|im_end|>
<|im_start|>assistant
```

Figure 11: An example AIME problem with the claims suffix appended to the user turn, rendered with the chat template exactly as it is fed to the model.

C PROGRAMMATIC CHECKER

We instantiate the grounded programmatic checker for each task, each of them sitting at difference position in the spectrum of informativeness.

Arithmetic For a completion c, let $E(c)$ be the set of parsed equations of the form x op $y = z$ that are grounded to the problem. For each $e \in E(c)$ define

$$\mathbf{1}e = \begin{cases} 1, & \text{if } x \text{ op } y \text{ evaluates to } z \text{ (within numeric tol.)}, \\ 0, & \text{otherwise.} \end{cases}$$

Then

$$r_{\text{py}}(c) = \begin{cases} \dfrac{1}{|E(c)|} \sum_{e \in E(c)} \mathbf{1}e, & |E(c)| > 0, \\ 0, & |E(c)| = 0, \end{cases} \qquad r\text{py}(c) \in [0, 1].$$

Countdown

$$r_{\text{valid}}(a) \;=\; \mathbf{1}[a \text{ parses}] \cdot \mathbf{1}[a \text{ uses the allowed numbers correctly}] \cdot \mathbf{1}[a \text{ evaluates to the target}]. \quad (8)$$

Every factor is computable from the source numbers and the target, both stated in the prompt.

Rubik's Cube Let $D(s)$ be the number of facelets of cube state s whose colour differs from the reference colour of their face, s_0 the scramble the prompt displays, and s_T the state after replaying the candidate's moves. We define

$$r_{\text{prog}}(a) \;=\; \mathbf{1}[\text{all moves of } a \text{ parse and are legal}] \cdot \max\left(0,\; \frac{D(s_0) - D(s_T)}{D(s_0)}\right). \quad (9)$$

s_0 is reconstructed by parsing the prompt's own ASCII rendering of the cube. On a $2 \times 2 \times 2$ cube there are no fixed centres, so a scramble such as `U D'` is a whole-cube rotation that the scorer accepts as solved; then $D(s_0) = 0$ and the ratio is undefined, and we return $\mathbf{1}[D(s_T) = 0]$ rather than the naive 0, which would mark correct answers wrong. The $\max(0, \cdot)$ floors answers that scramble the cube further, so that r_{prog} never pays for moving away from a solution.

LiveCodeBench Let G be group size, $n_{\text{chk}} = |T_h|$ the held-out test count, and for candidate i let p_i be its program. The programmatic checker is

$$r_{\text{chk}}(i) = \frac{1}{n_{\text{chk}}} \sum_{t \in T_h} \mathbf{1}\{\text{program } p_i \text{ passes } t\}.$$

In the implementation, it uses two stored input-output test cases for each training problem, still no reference solution and no evaluation tests, but it can be viewed as a weak form of external supervision rather than a purely prompt-derived checker.

AIME

$$r_{\text{trace}}(a) = \frac{1}{T} \sum_{t=1}^{T} \mathbf{1}\{\text{claim } t \text{ is executable and entailed by its cited premises}\}$$

where T is the number of claims in the emitted <claims> block.

D COVER HELPS ESCAPING WRONG MAJORITY

Problem: "Using 12, 10, 8, make 24". We have the rollout answers and their canonical classes in Table 9.

By mapping each answer to a canonical form, we get the 5 classes that are handed to the rewards in Table 10.

This is the case where we have unique plurality (4 votes), incorrect ($40 \neq 24$), and a correct class sits in the minority at $n = 1$.

#	Rollout's answer	Canonical class
g_0	`((12 - 8) * 10)`	`((12-8)*10)`
g_2	`(12 - 8) * 10`	`((12-8)*10)`
g_3	`((12-8)*10)`	`((12-8)*10)`
g_5	`(12 - 8) * 10`	`((12-8)*10)`
g_4	`(10 - 8) * 12`	`((10-8)*12)`
g_6	`12*2 + 10 - 8`	`(((12*2)+10)-8)`
g_1	*(no <answer> tag)*	`__nil_1__`
g_7	*(no <answer> tag)*	`__nil_7__`

Table 9: Problem: "Using 12, 10, 8, make 24". Rollout answers and their canonical classes.

Class	n	Correct	s_{yn}	s_{rec}	`consensus`	`abs_verify`	`cover`
`(12-8)*10` $= 40$	4	×	3.62	1.60	**4.00**	0.974	0.040
`(10-8)*12` $= 24$	1	✓	3.50	4.00	1.00	0.971	**0.891**
`12*2+10-8` (invents a 2)	1	×	4.38	1.71	1.00	**0.988**	0.010
`__nil_1__`	1	×	-6.50	-8.40	1.00	0.002	0.010
`__nil_7__`	1	×	-6.50	-8.40	1.00	0.002	0.010

Table 10: Deduplicated answer classes and the scores each reward assigns them. s_{yn}: the yesno mode, and the score is the logit margin at the next token, logit("Yes") − logit("No"). s_{rec}: the recompute mode. $\epsilon = 0.01$. `abs_verify` $= \sigma\big(s_{\text{yn}}(a)/\texttt{abs_temp}\big)$ with `abs_temp` $= 1.0$, $\sigma(x) = \frac{1}{1+e^{-x}}$.

- **`consensus`** → the argmax is the 4-vote class, so top1 $= \mathbf{0}$.
- **`abs_verify`** → the argmax is `12*2+10-8`, which the Yes/No verifier likes best (4.38) even though it uses a number it wasn't given, so top1 $= \mathbf{0}$.
- **`cover`** → re-derivation gives the correct class 4.00 vs. 1.60–1.71 for the others. The softmax at $T = 0.25$ concentrates mass on it ($\bar{v} \approx 0.88$), so 1×0.891 beats 4×0.040 and the 4-vote majority is overturned, so top1 $= \mathbf{1}$.

Averaging that 0/1 (fractional on ties) over all 365 problems gives the 0.00 / 0.14 / 0.69 bars.

Countdown Yes/No verification prompt

```
{problem}

Proposed final expression: {candidate}

Does it use each given number exactly once AND evaluate to the
target? Reply Yes or No.
```

Figure 12: Prompt used for Yes/No verification of a candidate expression.