Extended Abstract

Motivation. In this project, we study whether a language model can improve its answer at test time, without doing more training. The task we use is Countdown. In this task, the model is given several numbers and a target number, and it needs to produce an arithmetic expression that uses the given numbers and reaches the target. We choose this task because the answer is easy to verify by a program, but it is still not trivial for the model to generate the correct expression. The model needs to do several arithmetic steps and also follow the rule of using the numbers correctly. Therefore, Countdown gives us a relatively clean setting to test whether some failed model outputs can be repaired by feedback at inference time.

Method and Novelty. Our method is based on verifier-guided test-time repair. We start from a fixed base policy, trained with either RLOO or IPO. For each Countdown problem, we first sample several candidate answers from this policy. Then we use a deterministic Countdown verifier to check whether each answer has a valid format, uses the allowed numbers correctly, and evaluates to the target value. If at least one answer is correct, we count the problem as solved. If the initial samples fail, the repair loop takes failed drafts as repair candidates and sends them to a Qwen-based judge model. The judge gives a short natural-language critique about why the answer is wrong and what kind of correction may help. We then build a repair prompt using the original problem, the failed attempt, and the judge feedback, and sample again from the same frozen base policy.

The main difference from standard best-of-k sampling is that we do not only use the verifier as a final filter. In normal best-of-k, all samples are independent, and wrong samples are just discarded. In our method, a failed sample is used to create feedback, and the next attempt is conditioned on this feedback. Also, we do not update model weights in this process. The policy is frozen, and the improvement only comes from changing the inference-time context.

Implementation and Results. We implement the method as a recursive repair loop. The model first generates initial samples, and the rule-based verifier checks them. For unsolved problems, failed drafts are passed to the Qwen judge to get feedback. The model then tries again with a repair prompt, and the revised answers are checked by the same verifier. If the revised answers are still wrong, this process can be repeated for more recursive steps until the repair budget is used. In our experiments, we evaluate on 48 Countdown prompts and report pass@k for different values of k. We compare four main settings: RLOO, IPO, RLOO with repair, and IPO with repair.

In the current results, one repair step improves both policies. For RLOO, pass@32 increases from 0.3750 to 0.7500. For IPO, pass@32 increases from 0.2500 to 0.5208. The improvement also appears for smaller values of k, which means the feedback is useful even when the sampling budget is not very large. RLOO has better absolute performance than IPO in most cases. One possible reason is that RLOO is trained more directly with the rule-based reward, so its outputs are more aligned with the deterministic Countdown verifier. IPO starts from a weaker baseline, but it also improves a lot after repair. This suggests that many wrong IPO generations are not totally random mistakes. Some of them are close enough to be fixed after the model receives a useful critique.

Discussion, Limitations, and Conclusion. We also test different recursion depths. The result is not simply that more recursion is always better. A small number of repair steps is helpful, but deeper recursion gives mixed result. Sometimes the performance becomes flat, and sometimes it even goes down. One reason may be that the judge feedback is not always correct. Another reason is that after several rounds, the prompt becomes longer and may contain noisy or misleading information.

Overall, this experiment suggests that critique-conditioned resampling can repair some failed Countdown answers without retraining the model. The method is simple, but it changes the role of the verifier. The verifier is not only used to accept or reject final answers; together with the judge feedback, it becomes part of a test-time repair process. At the same time, the method has clear limitations. It requires extra judge calls and extra model samples, so the inference cost is higher. It also depends on the quality of the judge model and the prompt format. A natural next comparison would be to test against compute-matched best-of-k baselines, improve the judge prompt, and design better stopping rules for the repair loop. Even with these limitations, the results show that test-time repair is a useful direction for reasoning tasks where answers can be automatically checked.

Scaling Test-Time Computation for Countdown Reasoning through Verifier-Guided Resampling

Angel Meng Zhang
Department of Computer Science
Stanford University
angelzhm1@stanford.edu

Jiaxuan Sun
Department of Computer Science
Stanford University
jiaxuans@stanford.edu

Abstract

The performance of large language models has primarily improved through scaling data and training compute, but inference-time computation offers a complementary axis for enhancing reasoning. In this work, we study verifier-guided repair as a test-time mechanism for improving symbolic reasoning. Our method uses a Qwen-based judge model to generate natural-language critiques of failed Countdown solutions, which are then fed back into a frozen base policy to produce revised outputs. A deterministic verifier filters these outputs, forming an iterative feedback loop that improves generations without any parameter updates. We evaluate this approach on Countdown tasks using RLOO and IPO-trained policies. Results show that critique-conditioned resampling consistently improves pass@k, especially at shallow recursion depths, and that gains depend on the underlying policy's failure patterns. Overall, our findings suggest that verifier-guided test-time repair is a simple and effective form of inference-time scaling for symbolic reasoning.

1 Introduction

Recent advances in large language models have been driven primarily by scaling training data and compute. However, an emerging line of work suggests that model performance can also be significantly improved through additional computation at inference time, without updating model parameters. In particular, test-time scaling strategies—such as increasing sample diversity, applying verification, or extending reasoning steps—have shown promising gains in reasoning-intensive tasks.

Despite these advances, most existing approaches treat verification as a passive selection mechanism or rely on heuristic aggregation over independently sampled solutions. This limits the extent to which feedback from verification can actively shape subsequent reasoning trajectories.

In this work, we study whether verifier signals can be used not only for selection, but also as an active correction mechanism for improving symbolic reasoning. We hypothesize that verifier-guided natural-language feedback, when iteratively injected into a frozen policy via critique-conditioned resampling, can improve pass@k performance beyond standard test-time sampling and verification-based selection. Then, We use the Countdown task as a controlled setting for evaluating multi-step arithmetic reasoning, and investigate whether failed solutions can be systematically repaired at inference time using natural-language feedback.

Our contributions are:

- We propose a verifier-guided test-time repair framework that uses a Qwen judge model to generate natural-language critiques of failed Countdown solutions, and feeds this feedback back into the same frozen policy for critique-conditioned resampling.

- We show that this iterative repair procedure acts as a form of test-time scaling: without updating model weights, it can convert failed trajectories into verifier-correct solutions, with effectiveness varying across RLOO and IPO depending on their learned failure modes.
- Across our empirical results, verifier-guided repair consistently improves pass@k performance over baseline sampling, with the strongest gains at shallow repair depths and diminishing or non-monotonic returns as recursion depth increases.

2 Related Work

Recent work has increasingly focused on scaling test-time computation as a way to improve model reasoning without updating model parameters. Kwok et al. [1] show that scaling verification can be more effective than only scaling policy learning in vision-language-action alignment. Zhang et al. [2] study how reasoning models can allocate different amounts of thinking at test time. Setlur et al. [3] argue that scaling test-time compute without verification or RL is suboptimal. Kwok et al. [4] also show that increasing test-time sampling and verification can improve robustness in embodied reasoning settings. Lifshitz et al. [5] study multi-agent verification and show that using multiple verifiers can be another way to scale test-time computation.

These works show that inference-time computation and verification can improve reasoning performance. However, most of them use verification mainly for selection, scoring, or aggregation. In contrast, our method uses verifier-related feedback as an explicit repair signal. A Qwen judge critiques failed Countdown trajectories, and the critique is fed back into the same frozen policy for another attempt. Therefore, our method treats verification not only as a filter, but also as a way to guide the next generation.

3 Method

Our method uses a simple verifier-guided repair pipeline. At a high level, the base policy first generates candidate Countdown solutions. A deterministic verifier checks whether each solution is correct. If all initial candidates fail, a Qwen-based judge gives natural-language feedback on one failed response. This feedback is then inserted into a repair prompt, and the same frozen base policy is sampled again. The revised outputs are checked by the same verifier. This process can be repeated for a small number of recursive repair steps.

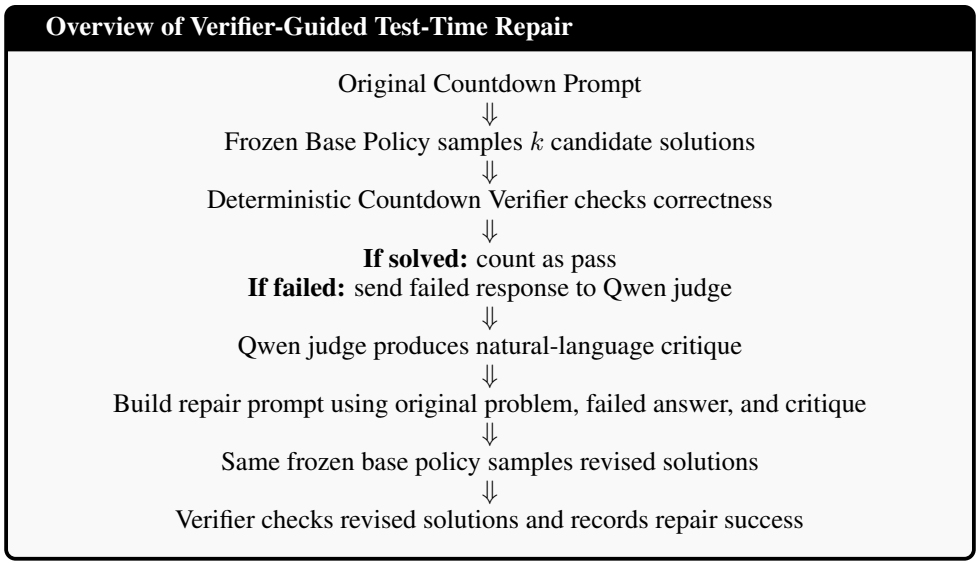

Figure 1: High-level pipeline of our verifier-guided repair method. The model weights are frozen throughout the process, and only the inference-time context is changed.

3.1 Algorithm and Technical Details

We use `Qwen/Qwen2.5-7B-Instruct` as the judge model to produce natural-language critiques for failed trajectories. Let x denote the original Countdown problem, y denote the target metadata, and r denote a response sampled from the fixed base policy π_θ, where the base policy is either the RLOO-trained or IPO-trained model. We use the deterministic Countdown verifier as the task checker. The verifier assigns a binary correctness score:

$$\tilde{s}_m = V(\tilde{r}_m, y).$$

$$V(r, y) \in \{0, 1\},$$

where $V(r, y) = 1$ if the extracted final expression both evaluates to the target value and uses the provided numbers according to the task constraints.

For each failed response r such that $V(r, y) = 0$, the judge produces a critique

$$j = J(x, r),$$

which describes why the response and provide some correction suggestions to . We then construct a critique-conditioned repair prompt

$$p_{\text{repair}} = \text{BuildRepairPrompt}(x, r, j),$$

containing the original problem, the failed response, the judge-generated critique, and an instruction to solve the problem again from scratch. Importantly, the judge's own proposed answer is not counted as a solution; it is used only as feedback for constructing the repair context.

The fixed base policy is then queried again using the repair prompt:

$$\{\tilde{r}_m\}_{m=1}^{M} \sim \pi_\theta(\cdot \mid p_{\text{repair}}),$$

where N is the same number of group size for revised samples. These revised attempts are therefore not independent samples from the original prompt; they are critique-conditioned samples generated by the same frozen policy.

For each revised response, we use the same determinant verifier to score. A repair attempt succeeds if at least one revised response is verifier-correct:

$$\text{RepairSuccess}(x) = \mathbb{I}\left[\max_{1 \le m \le M} V(\tilde{r}_m, y) = 1\right].$$

If all revised responses fail, we select the highest-scoring revised response as the next failed response ready to next recursive step,

$$m^\star = \arg\max_m V(\tilde{r}_m, y), \qquad r \leftarrow \tilde{r}_{m^\star},$$

and repeat the judge-an d-repair procedure up to a maximum recursion steps T. The model parameters θ remain fixed throughout this process; only the inference-time context is modified.

Algorithm 1 Verifier-Guided Recursive Countdown Repair

$n_{\text{judge}} \leftarrow 0$
for each prompt x in the evaluation set **do**
  Sample drafts $\{r_i\}_{i=1}^{k} \sim \pi_\theta(\cdot \mid x)$
  Compute scores $s_i = V(r_i, y)$
  INITIALPASS $\leftarrow \mathbf{1}[\max_i s_i = 1]$
  RECURSIVEPASS $\leftarrow$ INITIALPASS
  if not INITIALPASS **then**
    for each failed draft r **do**
      if $n_{\text{judge}} \geq$ `judge_max_failures` **then**
        break
      end if
      $c \leftarrow r$
      $t_{\text{global}} \leftarrow 0$
      while $t_{\text{global}} <$ `max_recursive_steps` **do**
        if $n_{\text{judge}} \geq$ `judge_max_failures` **then**
          break
        end if
        $t_{\text{global}} \leftarrow t_{\text{global}} + 1$
        $j \leftarrow \text{JUDGE}(x, c)$
        $n_{\text{judge}} \leftarrow n_{\text{judge}} + 1$
        Construct repair prompt
        $p_{\text{repair}} \leftarrow \text{BUILDREPAIRPROMPT}(x, c, j)$
        Sample revised responses $\{\tilde{r}_m\}_{m=1}^{M}$
        $\tilde{r}_m \sim \pi_\theta(\cdot \mid p_{\text{repair}})$
        Compute revised scores
        $\tilde{s}_m = V(\tilde{r}_m, y)$
        if $\max_m \tilde{s}_m = 1$ **then**
          RECURSIVEPASS $\leftarrow$ `true`
          break
        else
          $m^\star \leftarrow \arg\max_m \tilde{s}_m$
          $c \leftarrow \tilde{r}_{m^\star}$
        end if
      end while
      if RECURSIVEPASS **then**
        break
      end if
    end for
  end if
  Record INITIALPASS and RECURSIVEPASS
end for

3.2 The Novelty of Our Method Relative To Prior Work

Relative to prior work on test-time scaling, our contribution is to study verifier-guided repair as an explicit policy-improvement mechanism for symbolic reasoning, rather than only increasing sample count, verifier count, or reasoning length. Existing approaches show that more sampling, verification, or adaptive reasoning can improve performance, but they often treat verifier feedback as a selection signal or compute-allocation heuristic. In contrast, our method uses a concrete Qwen judge model to generate natural-language failure critiques, feeds those critiques back into the same frozen base policy, and iteratively resamples revised solutions that are accepted only by a deterministic task verifier. The novelty of our method makes the verifier not merely a selector but an inference-time feedback mechanism that reshapes the model's next attempt without updating weights.

4 Experimental Setup

4.1 Datasets and Baseline Model Training Details

We evaluate our method on the Countdown arithmetic reasoning task. In this task, each prompt gives a set of numbers and a target value, and the model needs to generate an arithmetic expression that uses the given numbers correctly and reaches the target. We use the same Countdown setting as in the default project pipeline. Our main baseline policies are the IPO-trained policy and the RLOO-trained policy. IPO is trained with a pairwise preference objective using the SFT model as the reference policy, while RLOO is trained with an online policy-gradient objective using the rule-based Countdown verifier as reward. In our milestone experiments, RLOO was trained for 100 optimization iterations, and we tracked RLOO loss, importance weight mean, KL loss, rollout accuracy, and final pass@k on the test set. The rollout accuracy was computed from sampled model generations during training, and the final checkpoint was evaluated with pass@k.

For the final extension evaluation, we use the trained IPO and RLOO checkpoints as frozen base policies. We do not update model parameters during verifier-guided repair. For each evaluation prompt, we first sample multiple candidate responses from the base policy and score them with the deterministic Countdown verifier. The verifier checks whether the final expression has the correct format, uses the provided numbers correctly, and evaluates to the target. We report pass@k for $k \in \{1, 2, 4, 8, 16, 32\}$ on 48 Countdown prompts. For failed initial responses, our method calls a Qwen-based judge model to generate natural-language feedback, builds a repair prompt, and samples revised responses from the same frozen base policy. We compare the original IPO and RLOO baselines with IPO + Ours and RLOO + Ours under the same verifier and pass@k evaluation. We use pass@k because Countdown answers are easy to verify automatically, and the metric directly measures whether at least one sampled response solves the problem.

4.2 Evaluation Metrics

For evaluation, we report pass@k. Given k initial samples $\{r_i\}_{i=1}^{k}$ for each prompt, the baseline pass@k metric is

$$\text{Pass@}k_{\text{base}} = \frac{1}{N} \sum_{n=1}^{N} \mathbb{I}\left[\max_{1 \le i \le k} V(r_{n,i}, y_n) = 1 \right],$$

where N is the number of evaluation prompts. For recursive repair, a prompt is counted as solved if either one of the initial k samples is correct or the repair procedure produces a verifier-correct revised response:

$$\text{Pass@}k_{\text{repair}} = \frac{1}{N} \sum_{n=1}^{N} \mathbb{I}\left[\max_{1 \le i \le k} V(r_{n,i}, y_n) = 1 \ \vee \ \text{RepairSuccess}(x_n) = 1 \right].$$

This metric measures whether critique-conditioned test-time computation can convert initially failed trajectories into verifier-correct solutions without updating the base model.

5 Experiment Results

5.1 Quantitative Evaluation

Table 1: Pass@k Comparison Across Methods on 48 Countdown Prompts (recursive step = 1)

k	SFT	RLOO	RLOO + Ours	IPO	IPO + Ours
1	0.0000	0.0417	0.4375	0.0208	0.2708
2	0.0000	0.0625	0.5625	0.0208	0.2917
4	0.0000	0.0833	0.5625	0.0417	0.3125
8	0.0200	0.1667	0.6875	0.0417	0.3125
16	0.0600	0.2708	0.7292	0.1458	0.4583
32	0.1000	0.3750	0.7500	0.2500	0.5208

Our experiment table shows that recursive repair substantially improves both RLOO and IPO across all pass@k values. RLOO + Ours achieves the strongest overall performance, reaching pass@32 = 0.7500 compared to the RLOO baseline 0.3750, while IPO + Ours improves IPO from 0.2500 to 0.5208 at pass@32. These results suggest that verifier-guided test-time repair consistently boosts reasoning accuracy, with large gains for both reward-trained and preference-trained policies.

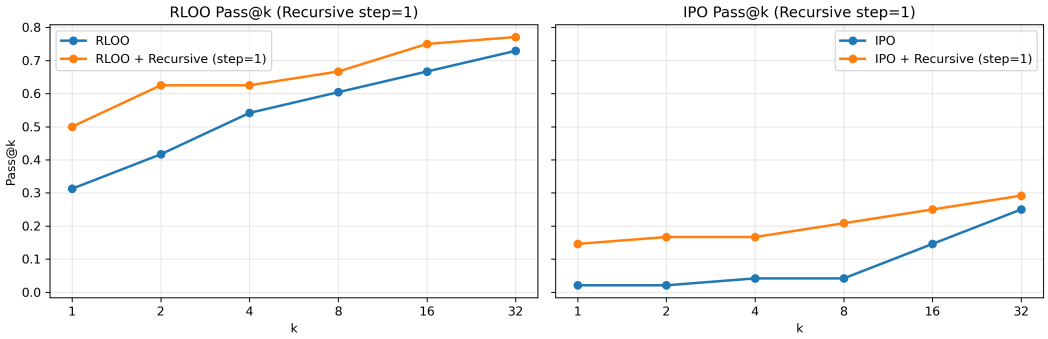

Figure 2: Figure 1 shows that one-step verifier-guided test-time repair improves pass@k performance for both RLOO and IPO across all evaluated sampling budgets. RLOO achieves higher absolute accuracy, while IPO exhibits a larger relative gain from repair, suggesting that verifier feedback is especially useful for weaker or less reward-aligned base policies.

We take recursive steps N = 1 as an example to compares baseline sampling against one-step verifier-guided repair for IPO and RLOO across pass@k values. In both cases, the recursive repair curve lies above the corresponding baseline curve, indicating that a single round of critique-conditioned regeneration improves Countdown accuracy without changing model weights. This supports the interpretation of the method as a test-time scaling strategy: additional inference-time computation, in the form of judge feedback and resampling, converts some initially failed generations into verifier-correct solutions.

The comparison also shows that RLOO and IPO benefit differently from repair. RLOO has substantially higher baseline performance across all k, suggesting that reward-based optimization produces generations that are already more aligned with the deterministic verifier. As a result, recursive repair yields a smaller but consistent improvement. IPO starts from a much weaker baseline, especially at low k, but one repair step produces a larger relative gain, indicating that verifier feedback is particularly useful when the base policy has more correctable errors. Overall, the figure suggests that verifier-guided repair complements pass@k sampling, with its largest relative benefit appearing for weaker or less verifier-aligned policies.

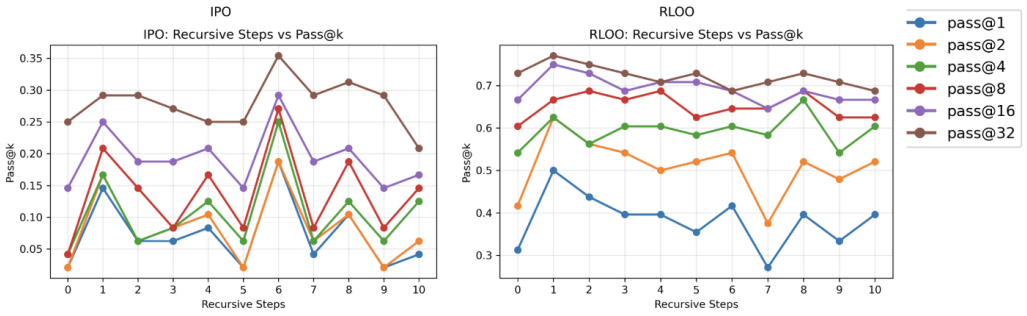

Figure 3: Pass@k accuracy across iterative repair depths for IPO and RLOO on the Countdown task. The curves show how verifier-guided test-time repair changes performance as the number of recursive repair steps increases, illustrating that additional inference-time computation can improve accuracy but does not yield monotonic gains at larger depths.

Moreover, we also conducted a quantitative experiment to show how IPO and RLOO respond to increasing amounts of verifier-guided test-time computation across different pass@k metrics. For both methods, performance generally improves from the baseline after a small number of repair recursive steps, indicating that critique-conditioned regeneration can recover some initially failed samples. However, the gains are not monotonic: after shallow repair recursive steps depths, additional repair recursive steps often produce flat or declining performance. This suggests that iterative repair is useful as a limited test-time intervention, but deeper recursion can introduce noisy feedback accumulation, prompt drift, or repeated invalid reasoning rather than consistent improvement. The comparison between IPO and RLOO also highlights different base-model behaviors. RLOO achieves higher absolute pass@k across most repair depths, suggesting that reward-based optimization produces a policy more aligned with the deterministic Countdown verifier. IPO, by contrast, starts from a weaker baseline but can show larger relative gains at early repair steps, implying that verifier feedback is especially helpful when the base policy has more correctable errors. Overall, the figure supports the view that verifier-guided repair acts as a form of test-time scaling, but its effectiveness depends on both the recursion budget and the quality of the underlying policy.

5.2 Qualitative Analysis

The original problem is:

> Using the numbers $[19, 25, 89]$, create an equation that equals 95. Use each number exactly once.

A baseline model sample may produce the following response:

Baseline Model Response

```
<think>
89 - 25 = 64.
64 + 19 = 83.
This is close to 95.
</think>

<answer>(89 - 25) + 19</answer>
```

The local deterministic verifier extracts the final expression and evaluates it:

$$(89 - 25) + 19 = 83.$$

Although the expression uses the provided numbers exactly once, it does not equal the target value 95. Therefore, the response receives a failing verifier score.

The failed response is then passed to the Qwen judge model. The judge identifies the arithmetic error and produces a critique suggestion such as:

Verifier/Judge Feedback

The expression uses the correct numbers exactly once, but it evaluates to 83 instead of 95. The model should try adding 25 to 89 and then subtracting 19.

The system constructs a critique-conditioned repair prompt for the fixed base policy:

Critique-Conditioned Repair Prompt

Previous failed attempt:
`<answer>(89 - 25) + 19</answer>`

Verifier/Judge critique:
The expression evaluates to 83 instead of 95. Try combining 89 and 25 first, then subtracting 19.

Now solve the original problem again from scratch. Use only the original numbers exactly once.

The baseline model is then sampled again using this augmented context. A revised response may be:

Repaired Base-Model Response

`<think>`
To reach 95, I can first add 89 and 25 to get 114.
Then subtract 19:
114 - 19 = 95.
`</think>`

`<answer>(89 + 25) - 19</answer>`

The verifier again evaluates the final expression:

$$(89 + 25) - 19 = 95.$$

This example shows how recursive repair can improve accuracy. Standard baseline sampling fails because it produces the incorrect expression $(89 - 25) + 19 = 83$. Recursive repair uses the failed trajectory to identify the arithmetic error, provides explicit corrective feedback, and prompts the original policy to retry under critique-conditioned context. Thus, the method can convert an initially failed sample into a verified correct solution without modifying the model weights.

6 Discussion

Limitations. While verifier-guided repair improves pass@k performance, its gains are not consistently monotonic with respect to recursive depth. In deeper repair steps, performance may plateau or degrade due to accumulated noise from iterative critique-conditioning, prompt drift, or over-reliance on imperfect judge feedback. Additionally, the approach depends on the quality of the external judge model, which may introduce systematic biases or incorrect critiques that misguide subsequent generations.

Broader impacts. This work highlights test-time computation as an alternative axis for improving reasoning without retraining models. Such approaches may reduce the need for expensive fine-tuning and enable more flexible deployment-time adaptation. However, increased reliance on external verifiers and judge models raises concerns about system complexity, reproducibility, and potential overfitting to verifier-specific biases rather than true task understanding.

Practical challenges. In practice, implementing recursive repair introduces non-trivial computational overhead due to repeated sampling and judge calls. We also observed sensitivity to prompt formatting and critique quality, which can significantly affect downstream repair success. Careful tuning of recursion depth and sampling budget is therefore necessary to balance performance gains with computational cost and stability.

7 Conclusion

In this work, we introduce a verifier-guided test-time repair method for symbolic reasoning. Our method first detects failed Countdown responses using a deterministic verifier, then uses a Qwen judge model to generate natural-language critiques of those failures. The critique is inserted into a repair prompt and passed back to the same frozen base policy, which resamples revised solutions until either a verifier-correct answer is found or the repair budget is exhausted. This procedure improves reasoning performance through additional inference-time computation alone, without updating the model weights. Our experiments on Countdown further compare how different post-training objectives, RLOO and IPO, respond to the same repair procedure, showing that test-time scaling interacts with the structure of the learned policy and can yield different marginal gains depending on the base model's failure modes .

8 Team Contributions

- **Group Member 1:** Angel Meng Zhang: Brainstorm the extension idea in proposal. Extension design and implementation. Write the final report. Summarize the final report into poster. Present the video of poster session.
- **Group Member 2:** Jiaxuan Sun: Implement SFT, RLOO and IPO and report milestone. Revised the final report and poster. Present the video of poster session.

Changes from Proposal The team split before final submission. All members agreed to receive different scores based on separated remaining deliverables, including the poster presentation (8% of the grade) and final project report (18% of the grade). The old team members include Angel Meng Zhang, Jiaxuan Sun, and Jingshu Liu; the new team consists of Angel Meng Zhang and Jiaxuan Sun in this final report. The model weights for this project has produced differently from milestone report due to the team splitting. After confirmation with our assigned TA, we are allowed to use current model weights for RLOO and IPO to submit this report as baseline model to expriment the extension idea without point deduction.

References

[1] Jacky Kwok, Xilun Zhang, Mengdi Xu, Yuejiang Liu, Azalia Mirhoseini, Chelsea Finn, and Marco Pavone. Scaling verification can be more effective than scaling policy learning for vision-language-action alignment, 2026.

[2] Junyu Zhang, Runpei Dong, Han Wang, Xuying Ning, Haoran Geng, Peihao Li, Xialin He, Yutong Bai, Jitendra Malik, Saurabh Gupta, and Huan Zhang. Alphaone: Reasoning models thinking slow and fast at test time, 2025.

[3] Amrith Setlur, Nived Rajaraman, Sergey Levine, and Aviral Kumar. Scaling test-time compute without verification or rl is suboptimal, 2025.

[4] Jacky Kwok, Christopher Agia, Rohan Sinha, Matt Foutter, Shulu Li, Ion Stoica, Azalia Mirhoseini, and Marco Pavone. Robomonkey: Scaling test-time sampling and verification for vision-language-action models, 2025.

[5] Shalev Lifshitz, Sheila A. McIlraith, and Yilun Du. Multi-agent verification: Scaling test-time compute with multiple verifiers, 2025.