Aletheia: What Makes RLVR For Code Verifiers Tick?

Anonymous ACL submission

Abstract

Multi-domain thinking verifiers trained via Reinforcement Learning from Verifiable Rewards (RLVR) are a prominent fixture of the Large Language Model (LLM) post-training pipeline, owing to their ability to robustly rate and rerank model outputs. However, the adoption of such verifiers towards code generation has been comparatively sparse, with execution feedback constituting the dominant signal. Nonetheless, code verifiers remain valuable toward judging model outputs in scenarios where execution feedback is hard to obtain and are a potentially powerful addition to the code generation post-training toolbox. To this end, we create and open-source **Aletheia**, a controlled testbed that enables execution-grounded evaluation of code verifiers' robustness across disparate policy models and covariate shifts. We examine components of the RLVR-based verifier training recipe widely credited for its success: (1) intermediate thinking traces, (2) learning from negative samples, and (3) on-policy training. While experiments show the optimality of RLVR, we uncover important opportunities to simplify the recipe. Particularly, despite code verification exhibiting positive training- and inference-time scaling, on-policy learning stands out as the key component at small verifier sizes, and thinking-based training emerges as the most important component at larger scales.[1]

1 Introduction

Reinforcement Learning (RL) for code generation typically relies on runtime signals to verify correctness (Le et al., 2022; Shojaee et al., 2023; Gehring et al., 2025; Liu et al., 2023a). However, self-contained executable codes with accompanying test-cases are a scarce resource—even for curated competitive programming datasets—and manual creation at scale is impractical (Wang et al., 2025c).

Prior work has attempted to sidestep this bottleneck via automating test-case generation (Li et al., 2022; Liu et al., 2023b; Chen et al., 2022; Li et al., 2023). However, these methods often struggle with test coverage and the inherent difficulty of specifying assertions for open-ended tasks. Alternatively, existing literature has dabbled in automating self-contained environment creation (Jain et al., 2025; Xie et al., 2024) and world modelling (Copet et al., 2025), which can be challenging for compiled languages without mature package managers.

In this work, we revisit yet another approach: surrogate code-execution verifiers (Ni et al., 2023; Li et al., 2025; Zeng et al., 2025; Shi et al., 2022; Zhang et al., 2023b). This approach involves training code verifier models to score code snippets on execution and auxiliary runtime outcomes without actually executing them. This allows post-training pipelines to leverage the code understanding and generalization capabilities of LLMs, providing a more flexible signal that obviates environment setup and code execution. Additionally, code verifiers afford feedback signals at multiple granularities, enabling long-horizon tasks where once-per-episode verifiable rewards are often too sparse to guide convergence (Cui et al., 2025).

Recently, general-domain thinking-enabled verifiers have proliferated in post-training pipelines for reasoning-heavy domains such as math and science (Liu et al., 2025b; Ma et al., 2025; Cen et al., 2025). This newfound dominance has been attained on the back of RLVR-based training (Chen et al., 2025d; Huang et al., 2025), unlocking improved interpretability (Gunjal et al., 2025) and robustness to reward hacking (Chen et al., 2025b). However, this paradigm has made fewer inroads in the field of code generation (Ni et al., 2024; Shum et al., 2025), where the few existing attempts are predominantly encoder-only regression models (Zhang et al., 2023b; Shi et al., 2022; Inala et al., 2022).

[1] We open source our dataset and codes at: https://anonymous.4open.science/r/arr2026-aletheia/

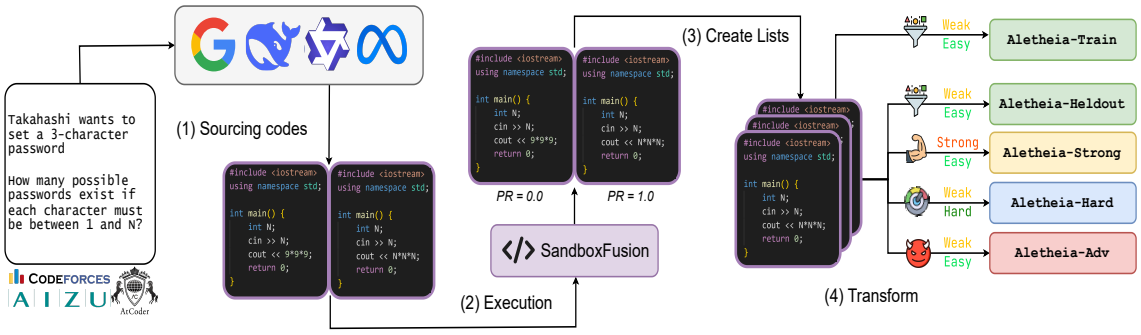

Figure 1: **The dataset creation pipeline.** We follow a three-step process by (1) sourcing competition-level programming questions from `CodeContests`$^+$ (Wang et al., 2025c) and sampling solutions from open-source LLMs, (2) executing the obtained solutions using SandboxFusion to obtain pass rates (PRs), (3) creating lists of 2-5 codes, of which exactly one is fully correct, and (4) dividing the resulting lists into our training and evaluation sets. A detailed description of the pipeline is in Section 2.1.

With this in view, we posit the question: **Is RLVR the optimal approach when training code verifiers?** An in-depth look into this question is interesting for multiple reasons. Firstly, prior work has demonstrated that small regression LMs are capable of approximating a plethora of execution outcomes (Akhauri et al., 2025). However, doing so in a manner robust to semantic-preserving or adversarial transforms is much more challenging (Haroon et al., 2025), even for the largest off-the-shelf models (Lyu et al., 2025). Secondly, the importance of reasoning in verification often strongly depends upon the difficulty of the problem (Pandit et al., 2025) and the quality of the generator being graded (Zhou et al., 2025). Finally, the conventional RLVR training recipe is compute-intensive and can suffer from low GPU utilization owing to the large number of models involved (the rollout policy, the behavior policy, and potentially an optional KL reference model, reward model, and a value model), the frequent parameter synchronizations between them, and the long rollout generations that are interspersed. Identifying efficient simplifications to the recipe could be key to unlocking the widespread use of surrogate code verifiers.

We thus formulate our study by identifying three major axes that contribute to the computational cost, and potentially to the success of RL – Generating long intermediate traces before converging to an answer (**Thinking**), learning from both positive and negative samples (**Negatives**), and learning from data generated by an updated policy (**Online**). Despite being characteristic of the RLVR recipe, the utility of all these axes is unclear, with arguments both for and against their effectiveness (detailed in Section 3). Examining the impact of these components enables the identification of simpler and more efficient training approaches, with broader applicability under computational constraints. Concretely, we summarize our contributions as:

- We create `Aletheia`, a controlled testbed to facilitate stress-testing code verifier training approaches across disparate policy generators and three realistic covariate shift settings (Section 2).
- We thoroughly ablate the relative contributions of reasoning, online learning, and learning from negatives in RLVR for verifier training. We conduct our analyses on verifiers ranging from 1.5B to 14B parameters in size (Section 3).
- We identify regimes where the optimal RLVR code verifier training recipe can be simplified without a marked loss in accuracy (Section 4).

2 Experimental Setup

2.1 Data Curation and Testbed Creation

We create a controlled training and evaluation setup to replicably evaluate the OOD capabilities of our trained verifiers. Code contest problems are particularly suitable for this purpose owing to their self-contained nature (enabling execution-grounded evaluation) and clearly annotated difficulty levels (enabling fine-grained analyses). To this end, we source programming problems from `CodeContests`$^+$ (Wang et al., 2025c), containing ≈ 25 synthetic test cases per instance. We filter out noisy instances where the synthetic testcases have a true positive rate or true negative rate < 0.9 against a pre-evaluated user-submitted solution pool. We also discard samples with fewer than 5 test cases and a time limit of more than 3 seconds. This gives us a dataset of 4903 programming questions with high-quality input-output test cases that can be solved

in any suitable programming language. We generate 50 completions each in Python, C++, and Java at a high sampling temperature of 1.0 using a mix of Weak and Strong LLMs as listed in Table 1. The strength of an LLM is determined based on its size and its standing on the BigCodeBench Leaderboard (Zhuo et al., 2024). The prompts for generating these codes are listed in Appendix E.

Model Family	Weak	Strong	ΔScore
DeepSeekCoder	6.7B	33B	6.5
Gemma2	9B	27B	5.1
Llama3.1	8B	70B	13.3
Qwen2.5Coder	7B	32B	8.6

Table 1: **Generators used for our datasets.** ΔScore is the difference between the Pass@1 BigCodeBench-Instruct scores of the Strong and Weak models. We use the -Instruct variant of all models in out experiments.

We then execute the code snippets using SandboxFusion[2] and calculate the percentage of test cases passed for each code (hereby referred to as the pass rate or PR). We construct lists containing $2-5$ solution codes, wherein each list contains codes with a distinct PRs and exactly one with PR $=1$. Since the space of valid outputs is discrete, creating lists of variable length disincentivizes models from gaming evaluation via random guessing. We divide these lists into Easy and Hard buckets based on their PRs; The Easy bucket contains lists where $\text{PR}_{\text{incorrect}} \in [0, 0.5]$ and Hard has $\text{PR}_{\text{incorrect}} \in [0.7, 0.9]$, making them partially correct and harder to distinguish from the correct code.

Algorithm	Thinking	Negatives	Online
GRPO-Think (§3)	✔	✔	✔
GRPO-Instruct (§3.1)	×	✔	✔
DPO (§3.2)	✔	✔	×
Batch-online GRPO (§3.2)	✔	✔	∼
RAFT (§3.3)	✔	×	✔

Table 2: **Overview of algorithms used.** "Thinking" denotes long intermediate trace generation, "Negatives" implies the use of negative samples, and "Online" indicates on-policy training.

We subsample the Weak-Easy bucket to $50,000$ instances, dubbed **Aletheia-Train**. Additionally, we create four evaluation datasets, equally distributed by list lengths and programming languages, and containing programming questions unseen during training (refer to Appendix A for statistics).

- **Aletheia-Heldout**. Easy comparisons by Weak models. This is our in-distribution baseline.
- **Aletheia-Strong**. Easy comparisons generated by Strong models. This dataset tests the robustness of the trained models to a shift in the generator's capability, without altering the quality of the codes being compared (Zhou et al., 2025).
- **Aletheia-Hard**. Hard comparisons generated by Weak models. Performance on this dataset reflects the easy-to-hard generalization abilities of our verifiers (Hase et al., 2024; Sun et al., 2024)
- **Aletheia-Adv**. To test the adversarial robustness of our verifiers, we apply three positive and negative modifications to the incorrect and correct codes in Aletheia-Heldout respectively,[3] based on prior work on biases in LLM judges (Lam et al., 2025; Hwang et al., 2025; Moon et al., 2025). We describe the modifications used in this dataset in detail in Appendix D.

2.2 Training details

We validate our findings across a wide range of model sizes, training 1.5B, 7B, and 14B parameter variants for each method. Unless explicitly mentioned, we initialize each method from the Deepseek-R1-Distill-Qwen2.5 models (DeepSeek-AI, 2025) because they have been warm-started to generate reasoning traces before answering. To ensure a fair comparison, all methods are trained for an identical number of gradient updates. For on-policy methods, we generate 16 responses at a high sampling temperature of 1.0 and award a $+1$ to generations that identify the correct candidate, and 0 otherwise. We provide a detailed description of our training setup and experiment with alternate rewards in Appendices B and C.

3 Research Questions and Results

We analyze the impact of three axes during training on downstream verifier performance – Thinking, Negatives, and Online, as described in Section 1. Across each ablation, we compare GRPO (Shao et al., 2025) to alternatives proposed in prior work, each lacking exactly one axis. The algorithms used in the subsequent sections are described in Table 2.

3.1 RQ1: Do verifiers need thinking traces?

Summary Of Findings 1

- Reasoning traces have a limited benefit for smaller models, but are vital to the success of large models.
- Increasing reasoning budgets (B_{tr}) trend similarly, having greater benefits for larger model sizes.
- Reasoning traces also allow verifiers to utilize inference-time compute effectively.

[2] https://github.com/bytedance/SandboxFusion

[3] Positive modifications are applied to all incorrect codes

Method	Size	B_{tr}	Per-Step-Cost ($)	Aletheia-Heldout	Aletheia-Strong	Aletheia-Hard	Aletheia-Adv	Average
GRPO-Instruct	1.5B	4096	0.587	38.78 ± 0.85	40.51 ± 0.85	31.22 ± 0.81	30.99 ± 0.33	35.41 ± 0.71
GRPO-Think		4096	1.208	42.73 ± 0.64	40.70 ± 0.61	36.32 ± 0.57	31.15 ± 0.25	37.76 ± 0.52
GRPO-Think		8192	2.491	46.82 ± 0.62	43.65 ± 0.59	41.61 ± 0.61	38.09 ± 0.25	42.55 ± 0.52
GRPO-Think		16384	7.806	49.58 ± 0.65	46.09 ± 0.62	40.74 ± 0.64	40.97 ± 0.26	44.38 ± 0.54
GRPO-Instruct	7B	4096	2.069	57.74 ± 0.88	51.80 ± 0.89	38.59 ± 0.87	52.20 ± 0.37	50.07 ± 0.75
GRPO-Think		4096	3.561	59.54 ± 0.71	55.00 ± 0.70	46.73 ± 0.74	44.04 ± 0.30	51.32 ± 0.61
GRPO-Think		8192	7.179	65.03 ± 0.57	56.96 ± 0.58	53.16 ± 0.66	52.03 ± 0.26	56.76 ± 0.52
GRPO-Think		16384	15.101	74.81 ± 0.57	67.28 ± 0.60	53.11 ± 0.69	65.04 ± 0.26	65.05 ± 0.53
GRPO-Instruct	14B	4096	9.463	63.45 ± 0.82	55.11 ± 0.84	44.15 ± 0.84	54.24 ± 0.35	54.26 ± 0.71
GRPO-Think		4096	8.558	73.23 ± 0.67	64.95 ± 0.71	54.56 ± 0.77	58.09 ± 0.31	62.69 ± 0.61
GRPO-Think		8192	14.900	78.37 ± 0.60	69.87 ± 0.65	61.74 ± 0.73	65.71 ± 0.29	68.91 ± 0.57
GRPO-Think		16384	36.992	88.02 ± 0.45	83.65 ± 0.49	66.84 ± 0.70	83.67 ± 0.21	80.54 ± 0.46

Table 3: **Results for ablating thinking.** We report `SC@1` scores and their 95% confidence interval. Thinking-style traces have little benefit for $1.5-7$B models, but are essential for the 14B model. Increasing B_{tr} has similar effects.

Background. Thinking traces are known to significantly boost their performance (Wei et al., 2022; Kojima et al., 2022). However, the source of these gains is ambiguous – with several works finding the lack of any causal relation between the model's CoT and final answer (Turpin et al., 2023; Wang et al., 2025b), casting doubt on the notion that the generated tokens allow the model to *think* before answering. This behaviour is less common, but still prominent in Large Reasoning Models (LRMs) (Chua and Evans, 2025). This indicates that *long* intermediate chains do not directly influence response quality (Stechly et al., 2025; Kambhampati et al., 2025), sparking an interest in generating shorter intermediate tokens (Arora and Zanette, 2025; Sui et al., 2025). In this section, we dig deeper to quantify the impact of greater thinking in improving verifiers by performing a controlled ablation.

Setup. We denote K as the number of generations sampled at inference time, and B_{tr} as the reasoning budget allowed during training (indirectly controlled by `max_completion_length` in `trl.GRPOTrainer`), and analyze the impact of varying both parameters on the final performance. We train four models, GRPO-Instruct with $B_{tr} = 4096$ and three GRPO-Think models with $B_{tr} \in [4096, 8192, 16384]$. We evaluate our models using self-consistency (Wang et al., 2023), and summarize our results in Table 3. Costs are calculated assuming $10.6 per H200-hour.[4]

Findings. GRPO-Think at $B_{tr} = 16384$ clearly outperforms all other algorithms across all model scales and evaluation settings, affirming the utility of long reasoning traces for verifier training (Chen et al., 2025d; Guo et al., 2025; Whitehouse et al., 2025). In contrast with existing work in other domains (Zhou et al., 2025), we find code verifiers are largely robust to changes in generator capability, with a mean performance drop of 4.85%, while easy-to-hard generalization is the hardest for most models, with an average drop of 14.2%. This robustness extends to policies much more capable than the verifier, signaling the potential value of verifiers in scalable supervision (Burns et al., 2024).

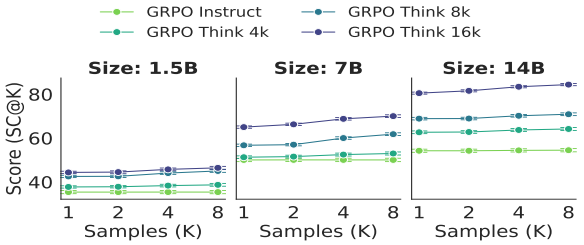

Figure 2: **Inference-time scaling for our thinking ablation.** Chain-of-Thought models do not benefit from additional compute, whereas thinking models see a small upward trend across all model sizes.

Small-medium models do not benefit much from the *style* of traces, evidenced by the small gap between the `-Instruct` and `-Think` models at $K = 1$ and $B_{tr} = 4096$. Interestingly, at the 14B scale, thinking traces become vital, significantly outperforming the traditional CoT baseline even at the same reasoning budget. An explanation for our findings is that thinking traces are close to the optimal prompt augmentation (Stechly et al., 2025; Kambhampati et al., 2025) for the 14B model, but not at the $1.5-7$B scale. Thus, we infer that the optimal prompt augmentation is a function of model size. Additionally, larger models are likely better at compositionally leveraging thinking primitives (Gandhi et al., 2025) in OOD settings.

While doubling B_{tr} from $4k$ to $8k$ yields reasonable robustness gains for the 1.5B model, further expansion to $16k$ results in diminishing returns. In contrast, the 7–14B models continue to benefit significantly from the $16k$ budget. It is known that large models are better at utilizing their context lengths (Hsieh et al., 2024; Liu et al., 2024). Thus, increasing the number of tokens generated helps

[4] https://docs.jarvislabs.ai/blog/h200-price

large models more, explaining their high scores. Increasing B_{tr} also makes the models more robust to shifts in generator capability and adversarial prompts. This is supported by prior work that finds reasoning models to be more robust to biases that LLMs are vulnerable to (Wang et al., 2025a).

We study the effect of increasing inference-time compute in Figure 2. Chain-of-thought models fail to utilize the extra compute, as seen by GRPO-Instruct, where SC@1 $\approx$ SC@8. However, even at the same B_{tr}, GRPO-Think sees a slight upward trend. This is in line with prior work that finds that reasoning traces allow models to utilize additional compute at test time (Lin et al., 2025).

3.2 RQ2: Is on-policy learning essential for verifier training?

Summary Of Findings 2

- While online training is generally more effective, its advantage diminishes with scale.
- Batch-online methods help small models, but afford only minor gains over offline methods with scale.
- Increasing inference-time compute always helps, but cannot bridge the online-offline performance gap.

Background. On-policy learning is perhaps the most widely studied and the most expensive aspect of RLVR training. Despite its effectiveness, on-policy training is very inefficient and often impractical. Thus, practitioners usually resort to introducing some amount of off-policyness to increase training efficiency (Noukhovitch et al., 2025; Piché et al., 2025). Moreover, there is no consensus on the necessity of on-policy learning. While some works find it vital to success in RL algorithms (Noukhovitch et al., 2025; Tang et al., 2024; Yu et al., 2025a), others claim that introducing a certain amount of off-policyness can match or even outperform fully on-policy methods (Lanchantin et al., 2025; Chen et al., 2025a; Song et al., 2024). In this section, we ablate the effects of on-policy learning on training code verifiers.

Setup. We study the impact of this decision through three representative algorithms. DPO-Think serves as our purely offline algorithm, and Batch-online GRPO represents the middle ground between online and offline methods, which samples a batch of responses and performs multiple gradient updates on mini-batches of the generated data. We detail the DPO dataset creation in Appendix B and present the results for this ablation in Table 4.

Findings. GRPO yet again dominates other methods, establishing the importance of fully on-policy learning in training verifiers. While the 1.5B DPO-Think significantly underperforms the batch-online and online models, the gap decreases with scale. For medium-large models, DPO-Think matches and even outperforms the other algorithms in some OOD evals, indicating that a well-curated offline dataset can yield impressive gains for verifier training. Despite requiring an offline dataset, the DPO methods are significantly cheaper than both GRPO methods, making them a suitable alternative at larger sizes. Moreover, the DPO dataset involves a non-recurring cost, in contrast to the continuous computational demands of online reinforcement learning. Our findings are supported by the fact that offline methods are effective when high-reward responses are likely under the training policy (Tajwar et al., 2024). The 1.5B model is highly unlikely to generate a high reward response, with an average SC@1 $\approx 1\%$, explaining DPO-Think's poor performance. The $7-14$B models have higher scores before training, and thus offline methods work well.

Batch-online GRPO fails to match GRPO, even underperforming DPO-Think in some OOD evaluation settings at larger sizes, and provides only a marginal cost saving compared to online GRPO. This contradicts the findings of Lanchantin et al. (2025), who found that batch-online training can even outperform fully online methods in verifiable math-based tasks. We explain this difference by the fact that they use `Llama-3.1-8B-Instruct`, which does not generate long reasoning chains (no Thinking), and is thus not directly comparable to our models. They also use the original GRPO recipe by Shao et al. (2024) with little modifications, while we incorporate improvements to the GRPO recipe, as detailed in Appendix B.

Figure 3: **Inference-time scaling plots for ablating on-policy learning.** Scaling inference-time compute benefits all models, but cannot fully replace the gains from on-policy learning.

Scaling inference compute benefits all methods and model sizes, as seen in Figure 3. Notably, the offline-online gap decreases at larger inference budgets, especially for the 1.5B model. While DPO-

Method	Size	Per-Step-Cost ($)	Aletheia-Heldout	Aletheia-Strong	Aletheia-Hard	Aletheia-Adv	Average
DPO-Think	1.5B	5.951*	21.70 ± 0.39	19.90 ± 0.35	23.41 ± 0.38	19.37 ± 0.15	21.11 ± 0.31
Batch-online GRPO		4.322	43.13 ± 0.55	39.89 ± 0.52	39.26 ± 0.54	33.83 ± 0.22	38.99 ± 0.46
GRPO-Think		7.806	49.58 ± 0.65	46.09 ± 0.62	40.74 ± 0.64	40.97 ± 0.26	44.38 ± 0.54
DPO-Think	7B	6.403*	63.75 ± 0.57	55.54 ± 0.57	51.20 ± 0.62	52.94 ± 0.25	55.88 ± 0.50
Batch-online GRPO		9.588	64.71 ± 0.61	56.18 ± 0.61	52.08 ± 0.66	49.35 ± 0.27	55.46 ± 0.54
GRPO-Think		15.101	74.81 ± 0.57	67.28 ± 0.60	53.11 ± 0.69	65.04 ± 0.26	65.05 ± 0.53
DPO-Think	14B	7.087*	82.56 ± 0.52	74.39 ± 0.58	67.58 ± 0.68	71.06 ± 0.26	73.89 ± 0.51
Batch-online GRPO		31.144	83.82 ± 0.50	76.33 ± 0.56	67.34 ± 0.68	73.45 ± 0.25	75.29 ± 0.50
GRPO-Think		36.992	88.02 ± 0.45	83.65 ± 0.49	66.84 ± 0.70	83.67 ± 0.21	80.54 ± 0.46

Table 4: **Results for ablating on-policy learning.** We report the SC@1 scores along with their 95% confidence interval. The gap between offline and online methods diminishes with scale. Batch-online methods can help the smallest models, but don't have an advantage for larger sizes. *DPO includes the cost for creating the offline dataset.

Think demonstrates consistent performance gains when scaling from $K = 1$ to $K = 8$, it fails to achieve parity with fully online GRPO. Thus, DPO-Think cannot be used as a lossless substitute for GRPO even at high inference budgets.

3.3 RQ3: Do negative samples benefit verifiers?

Summary Of Findings 3

- Learning from negative samples is equally beneficial across all model sizes and evaluation scenarios.
- Learning from only positive feedback is unstable; methods like RAFT underperform in OOD scenarios.
- Scaling inference compute cannot make up for the benefits of training on negative samples.

Background. Learning from negative samples is a characteristic of RL algorithms, as well as of contrastive learning methods like DPO (Rafailov et al., 2023), which optimize the RL objective directly. However, DPO suffers from the risk of reward over-optimization (Gao et al., 2023). Xu et al. (2024) find that even iterative DPO fails to beat the SFT baseline. Moreover, the importance of negative samples during training remains unclear. Arnal et al. (2025) find that learning successes are more important than learning from failures in an offline setup, while Zhu et al. (2025) find that negative reinforcement is much more important, even outperforming full training in some scenarios. Xiong et al. (2025) find that learning only from positives comes with a minor performance drop, and certain negative signals can even be detrimental. In this section, we quantify the effect of using negative samples for training verifiers.

Setup. We compare GRPO to a version of RAFT (Dong et al., 2023), modified to use verifiable rewards. RAFT works similarly to GRPO, where we sample and score N generations from an updated policy, but only trains on the correct responses using the next token prediction objective. We summarize our results in Table 5.

Findings. GRPO clearly outperforms RAFT across all model sizes and OOD scenarios, highlighting the importance of negative samples to training successful verifiers. Critically, the gap is constant across all model sizes, unlike in Online and Thinking, which are more important for small and large models, respectively (as described earlier). We plot the inference scaling trends in Figure 4. Similar to the other axes, increasing K does increase performance for all models. However, GRPO's single sample performance is still greater than RAFT's SC@8, proving that inference compute cannot make up for the lack of negative samples.

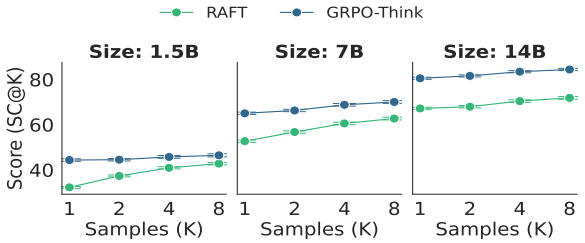

Figure 4: **Inference-time scaling plots for ablating negative samples.** The gap between RAFT and GRPO reduces with scaling compute, but persists.

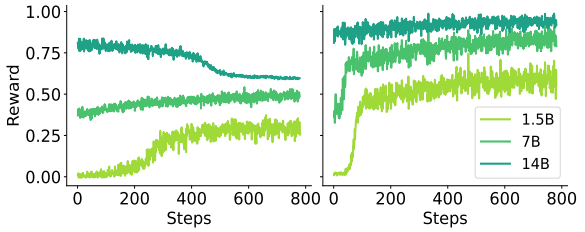

Figure 5: **Reward curves over training for RAFT and GRPO, respectively.** Training without negative samples becomes increasingly unstable with scale.

While the reward increases consistently for small models, RAFT significantly degrades due to overfitting for larger models, as seen in the reward curves for GRPO and RAFT in Figure 5. However, GRPO continues to improve even at the 14B scale, further emphasizing the importance of negative samples on stabilizing training. Despite RAFT offering up to 3× reduction in the cost per step, these savings do not justify the degradation for large models.

Method	Size	Per-Step-Cost ($)	Aletheia-Heldout	Aletheia-Strong	Aletheia-Hard	Aletheia-Adv	Average
RAFT	1.5B	4.167	34.76 ± 0.48	31.88 ± 0.44	33.67 ± 0.46	29.12 ± 0.19	32.30 ± 0.39
GRPO-Think		7.806	49.58 ± 0.65	46.09 ± 0.62	40.74 ± 0.64	40.97 ± 0.26	44.38 ± 0.54
RAFT	7B	6.948	60.86 ± 0.59	52.00 ± 0.58	48.84 ± 0.63	49.24 ± 0.25	52.72 ± 0.51
GRPO-Think		15.101	74.81 ± 0.57	67.28 ± 0.60	53.11 ± 0.69	65.04 ± 0.26	65.05 ± 0.53
RAFT	14B	12.906	75.55 ± 0.56	66.02 ± 0.61	65.23 ± 0.65	62.03 ± 0.27	67.20 ± 0.52
GRPO-Think		36.992	88.02 ± 0.45	83.65 ± 0.49	66.84 ± 0.70	83.67 ± 0.21	80.54 ± 0.46

Table 5: **Results for ablating negatives.** We report the `SC@1` scores along with their 95% confidence interval. Negative samples have an equal benefit for all model sizes.

4 Discussion and Takeaways

Having established the critical roles of the Thinking, Negatives, and Online axes in Section 3, and demonstrated GRPO's advantages over incomplete alternatives, we now translate these findings into actionable insights. This section provides actionable insights for practitioners optimizing verifier training during post-training.

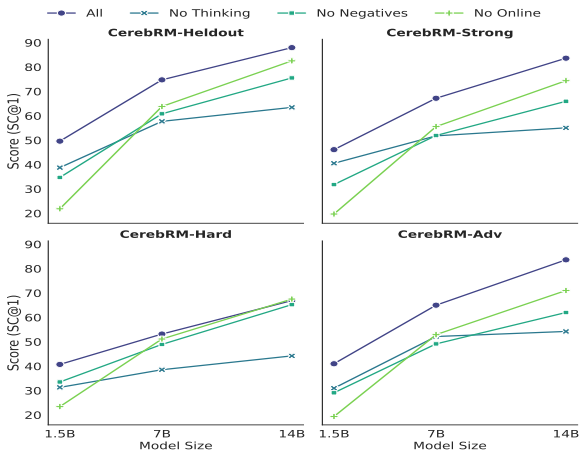

Figure 6: **Ablation of GRPO axes across model sizes.** While removing any axis reduces accuracy, the bottleneck shifts with scale. Online learning is crucial for small models (1.5B), whereas Thinking becomes the dominant factor for performance at larger scales (14B).

RLVR is the best method to train verifiers. Figure 6 shows that GRPO consistently matches or outperforms all ablated algorithms irrespective of model size and evaluation setting. This confirms that the three axes we study are not merely additive but synergistic, because removing any single component results in an inferior verifier. Moreover, GRPO performance increases linearly with model size in the `Easy-to-Hard` setting, suggesting that further scaling would yield significant gains. In other evaluations, the 14B model achieves over 80% accuracy, and further scaling would likely yield diminishing returns for a significant cost overhead.

Thinking is critical for `Easy-to-Hard` generalization. The incorrect codes in `Aletheia-Hard` pass $70-90\%$ of all test cases, compared to $< 50\%$ in the training data and other evaluation settings. This results in a greater similarity between the individual codes, as shown in Table 6, and thus increases the difficulty of comparison. Expectedly, there is a marked decrease in the performance of all algorithms in the `Aletheia-Hard` evaluations. Thinking traces become the most critical aspect of GRPO in this evaluation, with GRPO-Instruct being significantly worse than the other methods in the $7-14$B range. Thus, DPO-Think and RAFT are both suitable alternatives to GRPO for practitioners who expect their verifiers to encounter more difficult data than their training set.

On-policy learning is critical for small models, while thinking traces become important for large models. As seen in Figure 6, switching to a fully off-policy setup is catastrophic for the 1.5B model, with performance dropping an average of 23.4% across all evaluation settings. However, the $7-14$B models are surprisingly robust to offline data, even matching GRPO's performance in the `Easy-to-Hard` setting. The "No Thinking" ablation reveals another striking trend regarding model scale. At small scales (1.5B), the lack of thinking is the least critical, as its performance is the closest to that of the full GRPO algorithm. However, for the 14B model, the gap widens drastically – with GRPO-Instruct not improving from its 7B variant. This indicates that vanilla Chain-of-Thought prompting is unable to utilize the model's capacity beyond a certain point (around $7-8$B parameters), and thinking traces are required for further performance improvement. Practitioners training small verifiers may choose to eliminate thinking traces for faster training speeds and switch to offline learning for larger model sizes.

Learning exclusively from positive samples is consistently mediocre. While learning from negatives is not the most crucial aspect in any scenario, it is also not trivial and leads to significant gains across all model sizes and evaluations. This confirms the fact that learning from negative samples consistently increases performance across the scaling spectrum (Xiong et al., 2025; Zhu et al., 2025).

Scaling inference compute has little effect on verifier performance. The paradigm shift from discriminative Bradley-Terry reward models to generative verifiers is primarily driven by the potential to leverage test-time compute scaling. However, our experiments demonstrate that such scaling offers marginal performance gains and cannot serve as a drop-in replacement for fundamental improvements, such as increasing model size or utilizing advanced training algorithms like GRPO. We find that inference-time scaling primarily benefits models with a low base `SC@1`; while increased compute reduces the performance gap relative to GRPO-trained baselines, it fails to bridge it fully.

5 Related Work

We briefly elaborate on the three most relevant lines of existing work: (1) RLVR for verifier models, (2) surrogate code execution verifiers, and (3) prior analyses of RL in LLMs.

RLVR for LLM verifiers. Recent literature has significantly expanded verifier training by framing reward modeling as a verifiable re-ranking reasoning task (Whitehouse et al., 2025; Chen et al., 2025c; Huang et al., 2025). Such models have demonstrated state-of-the-art performance on popular reward model benchmarks and have been integrated into the production post-training pipelines of several modern LLMs (Chen et al., 2025b; Du et al., 2025; NVIDIA, 2025). Despite empirical gains, the optimal configuration for training such models remains under-explored. In this work, we uncover compute-optimal strategies for verifier training across three disparate model sizes by ablating three core components of the RLVR recipe.

Surrogate code execution. LLMs as surrogate code executors have taken several forms, including regression-based scoring models (Inala et al., 2022; Zhang et al., 2023b; Shi et al., 2022), natural language self-critique (Zhang et al., 2023a), and reasoning about compiler feedback (Chen et al., 2024). Alternatively, prior work has sought to train LLMs with execution semantics to directly (Ni et al., 2024) or indirectly (Copet et al., 2025; Ruan et al., 2025) improve their ability to abstractly reason about code execution. Beyond the file level, prior work has sought to reason about repository-level test-suite execution outcomes for software engineering tasks (Shum et al., 2025; Pan et al., 2025a). In this work, we hypothesize that the RLVR paradigm allows the training of robust code verifiers that can scalably supervise much larger policy models.

Prior analyses of RLVR in LLM training. Given the compute-intensive and inefficient nature of modern RL training (Noukhovitch et al., 2025; Piché et al., 2025), significant effort has been directed toward simplifying the RL objective (Dong et al., 2023; Rafailov et al., 2023; Agarwal et al., 2024; Hu et al., 2025). Our work complements these efforts by stripping down the core components of RLVR to identify avenues to make training verifiers more cost-effective.

It is unclear how important RLVR is to training successful code verifiers. The contribution of reasoning traces is debatable; while reasoning LLMs achieve benchmark-topping performance, Stechly et al. (2025); Kambhampati et al. (2025) demonstrate only a weak correlation between the correctness of intermediate traces and final answers, suggesting these gains may stem from alternative sources. Further decomposing the training signal, Xiong et al. (2025); Zhu et al. (2025) highlight the critical importance of negative samples in algorithms like GRPO, while Tajwar et al. (2024) delineate specific cases where on-policy learning is needed versus where off-policy samples suffice. Despite these insights, the field lacks a holistic assessment of how these components interact – leaving a critical gap in our understanding of what actually drives performance in RLVR training.

6 Conclusion

In this paper, we present an analysis of three different components of the RLVR recipe for verifier training and their contributions to its success – (1) generating long intermediate "thinking" traces, (2) learning from negative as well as positive samples, and (3) on-policy learning. We curate datasets to isolate the effects of training with the methods in our study, and evaluate the resulting verifiers on three out-of-distribution settings to ensure robustness. We find that GRPO is the most effective method for training verifiers across all model sizes, while the factors behind its success vary significantly. Smaller verifiers benefit the most from on-policy training, while larger models require thinking traces to reach their full potential. Future work can leverage our findings to judiciously allocate resources and reduce costs while training verifiers during post-training, depending on the size of their verifier, computation budget, target task distribution, and other relevant factors.

Limitations

Other RL algorithms and alternatives. Although we attempted to holistically and experimentally verify our RL training recipe, which we compare against other approaches, we stick with GRPO as it is most commonly used. Moreover, we represent each ablated axis with a single algorithm due to compute budget constraints, which adds some variance to our observations. Future work can build on and validate our findings by evaluating other RL algorithms like RLOO (Ahmadian et al., 2024) and REINFORCE++ (Hu et al., 2025), and also RL-inspired algorithms like ReST^{EM} (Singh et al., 2023), Iterative DPO (Xu et al., 2023; Xiong et al., 2024), NSR (Zhu et al., 2025), etc.

Differences in verifier training Our primary goal in this paper was to analyze the components of the RLVR recipe, rather than to train the best verifiers. As a result, the dataset we use to train our verifiers contains only easy comparisons from weak generators, which allowed us to systematically test the robustness of each algorithm to shifts in difficulty and generator capability. However, performance on this dataset saturates easily, especially at larger sizes (14B). We suggest that practitioners create their training dataset differently – to contain a mixture of all the evaluation scenarios that they anticipate encountering – to ensure strong performance as well as OOD robustness.

Restricted to coding problems Our work focuses on verifiers within the domain of competitive programming, a niche that remains relatively underexplored compared to standard reward modeling. Code contests offer a unique advantage for our research: they provide a high-precision mechanism for controlling *comparison difficulty* through granular testcase pass rates. Unlike task complexity – which measures the inherent difficulty of a problem – comparison difficulty quantifies the margin of error between solution attempts, allowing for a meticulous curation of candidates during training.

While constructing a controlled testbed such as ours in other domains remains non-trivial due to the absence of objective, fine-grained scoring, our findings remain broadly applicable. Our framework intentionally avoids code-specific heuristics or domain-dependent rewards; instead, it treats code execution purely as a high-fidelity ground truth. Instead, our reward function is a simple binary signal based on the verifier's ability to output a token corresponding to the correct candidate. Thus, our findings are relevant to practitioners from all domains, and future work can focus on validating our findings in other domains of verifier training.

Ethical Considerations

Although we focus on validating the robustness of our verifiers in several out-of-distribution scenarios, including code snippets generated by a potential adversary, we do not cover the full space of possible codes in our training or evaluation. Thus, it is possible our verifiers are susceptible to reward hacking – incorrectly assigning high scores to incorrect and potentially harmful responses. We take steps to mitigate these risks by thoroughly documenting our entire workflow. We also plan on making our datasets, codes, and models open-source under the `CC BY-NC-SA 4.0` License.

References

Rishabh Agarwal, Nino Vieillard, Yongchao Zhou, Piotr Stanczyk, Sabela Ramos Garea, Matthieu Geist, and Olivier Bachem. 2024. On-policy distillation of language models: Learning from self-generated mistakes. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

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.

Yash Akhauri, Xingyou Song, Arissa Wongpanich, Bryan Lewandowski, and Mohamed S. Abdelfattah. 2025. Regression language models for code. *CoRR*, abs/2509.26476.

Charles Arnal, GaĂŤtan Narozniak, Vivien Cabannes, Yunhao Tang, Julia Kempe, and Remi Munos. 2025. Asymmetric reinforce for off-policy reinforcement learning: Balancing positive and negative rewards. *ArXiv preprint*, abs/2506.20520.

Daman Arora and Andrea Zanette. 2025. Training language models to reason efficiently.

Anirudh Bharadwaj, Chaitanya Malaviya, Nitish Joshi, and Mark Yatskar. 2025. Flattery, Fluff, and Fog: Diagnosing and Mitigating Idiosyncratic Biases in Preference Models.

Collin Burns, Pavel Izmailov, Jan Hendrik Kirchner, Bowen Baker, Leo Gao, Leopold Aschenbrenner, Yining Chen, Adrien Ecoffet, Manas Joglekar, Jan Leike, Ilya Sutskever, and Jeffrey Wu. 2024. Weak-to-strong generalization: Eliciting strong capabilities with weak supervision. In *Forty-first International Conference*

on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024. OpenReview.net.

Zhepeng Cen, Haolin Chen, Shiyu Wang, Zuxin Liu, Zhiwei Liu, Ding Zhao, Silvio Savarese, Caiming Xiong, Huan Wang, and Weiran Yao. 2025. Webscale-rl: Automated data pipeline for scaling RL data to pretraining levels. *CoRR*, abs/2510.06499.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*.

Howard Chen, Noam Razin, Karthik Narasimhan, and Danqi Chen. 2025a. Retaining by doing: The role of on-policy data in mitigating forgetting. *ArXiv preprint*, abs/2510.18874.

Jiaze Chen, Tiantian Fan, Xin Liu, Lingjun Liu, Zhiqi Lin, Mingxuan Wang, Chengyi Wang, Xiangpeng Wei, Wenyuan Xu, Yufeng Yuan, Yu Yue, Lin Yan, Qiying Yu, Xiaochen Zuo, Chi Zhang, Ruofei Zhu, Zhecheng An, Zhihao Bai, Yu Bao, and 80 others. 2025b. Seed1.5-thinking: Advancing superb reasoning models with reinforcement learning. *ArXiv preprint*, abs/2504.13914.

Nuo Chen, Zhiyuan Hu, Qingyun Zou, Jiaying Wu, Qian Wang, Bryan Hooi, and Bingsheng He. 2025c. Judgelrm: Large reasoning models as a judge. *ArXiv preprint*, abs/2504.00050.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching large language models to self-debug. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Xiusi Chen, Gaotang Li, Ziqi Wang, Bowen Jin, Cheng Qian, Yu Wang, Hongru Wang, Yu Zhang, Denghui Zhang, Tong Zhang, Hanghang Tong, and Heng Ji. 2025d. RM-R1: Reward Modeling as Reasoning.

James Chua and Owain Evans. 2025. Are deepseek r1 and other reasoning models more faithful?

Jade Copet, Quentin Carbonneaux, Gal Cohen, Jonas Gehring, Jacob Kahn, Jannik Kossen, Felix Kreuk, Emily McMilin, Michel Meyer, Yuxiang Wei, David Zhang, Kunhao Zheng, Jordi Armengol-Estapé, Pedram Bashiri, Maximilian Beck, Pierre Chambon, Abhishek Charnalia, Chris Cummins, Juliette Decugis, and 31 others. 2025. CWM: an open-weights LLM for research on code generation with world models. *CoRR*, abs/2510.02387.

Ganqu Cui, Lifan Yuan, Zefan Wang, Hanbin Wang, Wendi Li, Bingxiang He, Yuchen Fan, Tianyu Yu, Qixin Xu, Weize Chen, Jiarui Yuan, Huayu Chen, Kaiyan Zhang, Xingtai Lv, Shuo Wang, Yuan Yao, Xu Han, Hao Peng, Yu Cheng, and 4 others. 2025. Process reinforcement through implicit rewards. *ArXiv preprint*, abs/2502.01456.

DeepSeek-AI. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning.

Hanze Dong, Wei Xiong, Deepanshu Goyal, Yihan Zhang, Winnie Chow, Rui Pan, Shizhe Diao, Jipeng Zhang, KaShun SHUM, and Tong Zhang. 2023. RAFT: Reward ranked finetuning for generative foundation model alignment. *Transactions on Machine Learning Research*.

Angang Du, Bofei Gao, Bowei Xing, Changjiu Jiang, Cheng Chen, Cheng Li, Chenjun Xiao, Chenzhuang Du, Chonghua Liao, and 1 others. 2025. Kimi k1. 5: Scaling reinforcement learning with llms. *ArXiv preprint*, abs/2501.12599.

Kenneth Enevoldsen, Isaac Chung, Imene Kerboua, Márton Kardos, Ashwin Mathur, David Stap, Jay Gala, Wissam Siblini, Dominik Krzemiński, Genta Indra Winata, Saba Sturua, Saiteja Utpala, Mathieu Ciancone, Marion Schaeffer, Gabriel Sequeira, Diganta Misra, Shreeya Dhakal, Jonathan Rystrøm, Roman Solomatin, and 67 others. 2025. Mmteb: Massive multilingual text embedding benchmark. *ArXiv preprint*, abs/2502.13595.

Kanishk Gandhi, Ayush Chakravarthy, Anikait Singh, Nathan Lile, and Noah D. Goodman. 2025. Cognitive behaviors that enable self-improving reasoners, or, four habits of highly effective stars. *ArXiv preprint*, abs/2503.01307.

Leo Gao, John Schulman, and Jacob Hilton. 2023. Scaling laws for reward model overoptimization. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 10835–10866. PMLR.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. 2025. RLEF: grounding code llms in execution feedback with reinforcement learning. In *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*. OpenReview.net.

Alexey Gorbatovski, Boris Shaposhnikov, Alexey Malakhov, Nikita Surnachev, Yaroslav Aksenov, Ian Maksimov, Nikita Balagansky, and Daniil Gavrilov. 2024. Learn your reference model for real good alignment. *ArXiv preprint*, abs/2404.09656.

Anisha Gunjal, Anthony Wang, Elaine Lau, Vaskar Nath, Bing Liu, and Sean Hendryx. 2025. Rubrics as rewards: Reinforcement learning beyond verifiable domains. *CoRR*, abs/2507.17746.

Jiaxin Guo, Zewen Chi, Li Dong, Qingxiu Dong, Xun Wu, Shaohan Huang, and Furu Wei. 2025. Reward reasoning model.

Sabaat Haroon, Ahmad Faraz Khan, Ahmad Humayun, Waris Gill, Abdul Haddi Amjad, Ali R. Butt, Mohammad Taha Khan, and Muhammad Ali Gulzar. 2025. How accurately do large language models understand code? *CoRR*, abs/2504.04372.

Peter Hase, Mohit Bansal, Peter Clark, and Sarah Wiegreffe. 2024. The Unreasonable Effectiveness of Easy Training Data for Hard Tasks.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. 2024. Ruler: What's the real context size of your long-context language models? *ArXiv preprint*, abs/2404.06654.

Jian Hu, Jason Klein Liu, Haotian Xu, and Wei Shen. 2025. Reinforce++: Stabilizing critic-free policy optimization with global advantage normalization.

Hui Huang, Yancheng He, Hongli Zhou, Rui Zhang, Wei Liu, Weixun Wang, Wenbo Su, Bo Zheng, and Jiaheng Liu. 2025. Think-j: Learning to think for generative llm-as-a-judge. *ArXiv preprint*, abs/2505.14268.

Yerin Hwang, Dongryeol Lee, Taegwan Kang, Yongil Kim, and Kyomin Jung. 2025. Can You Trick the Grader? Adversarial Persuasion of LLM Judges.

Jeevana Priya Inala, Chenglong Wang, Mei Yang, Andrés Codas, Mark Encarnación, Shuvendu K. Lahiri, Madanlal Musuvathi, and Jianfeng Gao. 2022. Fault-aware neural code rankers. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. 2025. R2e-gym: Procedural environments and hybrid verifiers for scaling open-weights SWE agents. *CoRR*, abs/2504.07164.

Subbarao Kambhampati, Kaya Stechly, Karthik Valmeekam, Lucas Saldyt, Siddhant Bhambri, Vardhan Palod, Atharva Gundawar, Soumya Rani Samineni, Durgesh Kalwar, and Upasana Biswas. 2025. Stop Anthropomorphizing Intermediate Tokens as Reasoning/Thinking Traces!

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Man Ho Lam, Chaozheng Wang, Jen-tse Huang, and Michael R. Lyu. 2025. CodeCrash: Stress Testing LLM Reasoning under Structural and Semantic Perturbations.

Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V. Miranda, Alisa Liu, Nouha Dziri, Shane Lyu, Yuling Gu, Saumya Malik, Victoria Graf, Jena D. Hwang, Jiangjiang Yang, Ronan Le Bras, Oyvind Tafjord, Chris Wilhelm, Luca Soldaini, and 4 others. 2024. Tulu 3: Pushing Frontiers in Open Language Model Post-Training.

Jack Lanchantin, Angelica Chen, Janice Lan, Xian Li, Swarnadeep Saha, Tianlu Wang, Jing Xu, Ping Yu, Weizhe Yuan, Jason E. Weston, Sainbayar Sukhbaatar, and Ilia Kulikov. 2025. Bridging Offline and Online Reinforcement Learning for LLMs.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu-Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Qingyao Li, Xinyi Dai, Xiangyang Li, Weinan Zhang, Yasheng Wang, Ruiming Tang, and Yong Yu. 2025. Codeprm: Execution feedback-enhanced process reward model for code generation. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 8169–8182.

Rongao Li, Jie Fu, Bo-Wen Zhang, Tao Huang, Zhihong Sun, Chen Lyu, Guang Liu, Zhi Jin, and Ge Li. 2023. Taco: Topics in algorithmic code generation dataset. *arXiv preprint arXiv:2312.14852*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, and 7 others. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Junhong Lin, Xinyue Zeng, Jie Zhu, Song Wang, Julian Shun, Jun Wu, and Dawei Zhou. 2025. Plan and budget: Effective and efficient test-time scaling on large language model reasoning. *CoRR*, abs/2505.16122.

Jiate Liu, Yiqin Zhu, Kaiwen Xiao, Qiang Fu, Xiao Han, Wei Yang, and Deheng Ye. 2023a. RLTF: reinforcement learning from unit test feedback. *Trans. Mach. Learn. Res.*, 2023.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. 2023b. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, volume 36, pages 21558–21572. Curran Associates, Inc.

Mingjie Liu, Shizhe Diao, Ximing Lu, Jian Hu, Xin Dong, Yejin Choi, Jan Kautz, and Yi Dong. 2025a. ProRL: Prolonged Reinforcement Learning Expands Reasoning Boundaries in Large Language Models.

Shudong Liu, Hongwei Liu, Junnan Liu, Linchen Xiao, Songyang Gao, Chengqi Lyu, Yuzhe Gu, Wenwei Zhang, Derek F. Wong, Songyang Zhang, and Kai Chen. 2025b. CompassVerifier: A unified and robust verifier for LLMs evaluation and outcome reward. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages

33466–33494, Suzhou, China. Association for Computational Linguistics.

Xiang Liu, Peijie Dong, Xuming Hu, and Xiaowen Chu. 2024. Longgenbench: Long-context generation benchmark. *ArXiv preprint*, abs/2410.04199.

Zihe Liu, Jiashun Liu, Yancheng He, Weixun Wang, Jiaheng Liu, Ling Pan, Xinyu Hu, Shaopan Xiong, Ju Huang, Jian Hu, Shengyi Huang, Siran Yang, Jiamang Wang, Wenbo Su, and Bo Zheng. 2025c. Part I: Tricks or Traps? A Deep Dive into RL for LLM Reasoning.

Zijun Liu, Peiyi Wang, Runxin Xu, Shirong Ma, Chong Ruan, Peng Li, Yang Liu, and Yu Wu. 2025d. Inference-time scaling for generalist reward modeling.

Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

Bohan Lyu, Siqiao Huang, and Zichen Liang. 2025. SURGE: on the potential of large language models as general-purpose surrogate code executors. *CoRR*, abs/2502.11167.

Xueguang Ma, Qian Liu, Dongfu Jiang, Ge Zhang, Zejun Ma, and Wenhu Chen. 2025. General-reasoner: Advancing LLM reasoning across all domains. *CoRR*, abs/2505.14652.

Jiwon Moon, Yerin Hwang, Dongryeol Lee, Taegwan Kang, Yongil Kim, and Kyomin Jung. 2025. Don't Judge Code by Its Cover: Exploring Biases in LLM Judges for Code Evaluation.

Niklas Muennighoff, Nouamane Tazi, Loic Magne, and Nils Reimers. 2023. MTEB: Massive text embedding benchmark. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*, pages 2014–2037, Dubrovnik, Croatia. Association for Computational Linguistics.

Ansong Ni, Miltiadis Allamanis, Arman Cohan, Yinlin Deng, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2024. Next: Teaching large language models to reason about code execution. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. 2023. Lever: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning*, pages 26106–26128. PMLR.

Michael Noukhovitch, Shengyi Huang, Sophie Xhonneux, Arian Hosseini, Rishabh Agarwal, and Aaron C. Courville. 2025. Asynchronous RLHF: faster and more efficient off-policy RL for language models. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

NVIDIA. 2025. Nemotron 3 Nano: Open, efficient mixture-of-experts hybrid Mamba-Transformer model for Agentic reasoning. Technical report.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. 2025a. Training software engineering agents and verifiers with swe-gym. In *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*. OpenReview.net.

Yu Pan, Zhongze Cai, Guanting Chen, Huaiyang Zhong, and Chonghuan Wang. 2025b. What Matters in Data for DPO?

Shrey Pandit, Austin Xu, Xuan-Phi Nguyen, Yifei Ming, Caiming Xiong, and Shafiq Joty. 2025. Hard2verify: A step-level verification benchmark for open-ended frontier math. *CoRR*, abs/2510.13744.

Indraneil Paul, Haoyi Yang, Goran Glavaš, Kristian Kersting, and Iryna Gurevych. 2025. Obscuracoder: Powering efficient code lm pre-training via obfuscation grounding. *ArXiv preprint*, abs/2504.00019.

Alexandre Piché, Ehsan Kamalloo, Rafael Pardinas, Xiaoyin Chen, and Dzmitry Bahdanau. 2025. Pipelinerl: Faster on-policy reinforcement learning for long sequence generation. *ArXiv preprint*, abs/2509.19128.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D. Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*, pages 3505–3506. ACM.

Chi Ruan, Dongfu Jiang, Yubo Wang, and Wenhu Chen. 2025. Critique-coder: Enhancing coder models by critique reinforcement learning. *CoRR*, abs/2509.22824.

Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. 2024. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.

Rulin Shao, Shuyue Stella Li, Rui Xin, Scott Geng, Yiping Wang, Sewoong Oh, Simon Shaolei Du, Nathan Lambert, Sewon Min, Ranjay Krishna, Yulia Tsvetkov, Hannaneh Hajishirzi, Pang Wei Koh, and Luke Zettlemoyer. 2025. Spurious rewards: Rethinking training signals in rlvr.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models.

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I. Wang. 2022. Natural language to code translation with execution. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, pages 3533–3546. Association for Computational Linguistics.

Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K. Reddy. 2023. Execution-based code generation using deep reinforcement learning. *Preprint*, arXiv:2301.13816.

KaShun Shum, Binyuan Hui, Jiawei Chen, Lei Zhang, W. X., Jiaxin Yang, Yuzhen Huang, Junyang Lin, and Junxian He. 2025. Swe-rm: Execution-free feedback for software engineering agents.

Avi Singh, John D. Co-Reyes, Rishabh Agarwal, Ankesh Anand, Piyush Patil, Xavier Garcia, Peter J. Liu, James Harrison, Jaehoon Lee, Kelvin Xu, Aaron Parisi, Abhishek Kumar, Alex Alemi, Alex Rizkowsky, Azade Nova, Ben Adlam, Bernd Bohnet, Gamaleldin Elsayed, Hanie Sedghi, and 22 others. 2023. Beyond Human Data: Scaling Self-Training for Problem-Solving with Language Models.

Yuda Song, Gokul Swamy, Aarti Singh, J. Andrew Bagnell, and Wen Sun. 2024. The importance of online data: Understanding preference fine-tuning via coverage. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.

Kaya Stechly, Karthik Valmeekam, Atharva Gundawar, Vardhan Palod, and Subbarao Kambhampati. 2025. Beyond semantics: The unreasonable effectiveness of reasonless intermediate tokens. *ArXiv preprint*, abs/2505.13775.

Yang Sui, Yu-Neng Chuang, Guanchu Wang, Jiamu Zhang, Tianyi Zhang, Jiayi Yuan, Hongyi Liu, Andrew Wen, Shaochen Zhong, Na Zou, Hanjie Chen, and Xia Hu. 2025. Stop overthinking: A survey on efficient reasoning for large language models.

Zhiqing Sun, Longhui Yu, Yikang Shen, Weiyang Liu, Yiming Yang, Sean Welleck, and Chuang Gan. 2024. Easy-to-hard generalization: Scalable alignment beyond human supervision. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.

Fahim Tajwar, Anikait Singh, Archit Sharma, Rafael Rafailov, Jeff Schneider, Tengyang Xie, Stefano Ermon, Chelsea Finn, and Aviral Kumar. 2024. Preference fine-tuning of llms should leverage suboptimal, on-policy data. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

Yunhao Tang, Daniel Zhaohan Guo, Zeyu Zheng, Daniele Calandriello, Yuan Cao, Eugene Tarassov, Rémi Munos, Bernardo Ávila Pires, Michal Valko, Yong Cheng, and 1 others. 2024. Understanding the performance gap between online and offline alignment algorithms. *ArXiv preprint*, abs/2405.08448.

Miles Turpin, Julian Michael, Ethan Perez, and Samuel R. Bowman. 2023. Language models don't always say what they think: Unfaithful explanations in chain-of-thought prompting. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Qian Wang, Zhanzhi Lou, Zhenheng Tang, Nuo Chen, Xuandong Zhao, Wenxuan Zhang, Dawn Song, and Bingsheng He. 2025a. Assessing Judging Bias in Large Reasoning Models: An Empirical Study.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Yanbo Wang, Yongcan Yu, Jian Liang, and Ran He. 2025b. A comprehensive survey on trustworthiness in reasoning with large language models.

Zihan Wang, Siyao Liu, Yang Sun, Hongyan Li, and Kai Shen. 2025c. Codecontests+: High-quality test case generation for competitive programming. *ArXiv preprint*, abs/2506.05817.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Chenxi Whitehouse, Tianlu Wang, Ping Yu, Xian Li, Jason Weston, Ilia Kulikov, and Swarnadeep Saha. 2025. J1: incentivizing thinking in llm-as-a-judge via reinforcement learning. *ArXiv preprint*, abs/2505.10320.

Yiqing Xie, Alex Xie, Divyanshu Sheth, Pengfei Liu, Daniel Fried, and Carolyn P. Rosé. 2024. Codebenchgen: Creating scalable execution-based code generation benchmarks. *CoRR*, abs/2404.00566.

Wei Xiong, Hanze Dong, Chenlu Ye, Ziqi Wang, Han Zhong, Heng Ji, Nan Jiang, and Tong Zhang. 2024. Iterative preference learning from human feedback: Bridging theory and practice for RLHF under kl-constraint. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

Wei Xiong, Jiarui Yao, Yuhui Xu, Bo Pang, Lei Wang, Doyen Sahoo, Junnan Li, Nan Jiang, Tong Zhang, Caiming Xiong, and Hanze Dong. 2025. A Minimalist Approach to LLM Reasoning: From Rejection Sampling to Reinforce.

Jing Xu, Andrew Lee, Sainbayar Sukhbaatar, and Jason Weston. 2023. Some things are more cringe than others: Iterative preference optimization with the pairwise cringe loss. *ArXiv preprint*, abs/2312.16682.

Shusheng Xu, Wei Fu, Jiaxuan Gao, Wenjie Ye, Weilin Liu, Zhiyu Mei, Guangju Wang, Chao Yu, and Yi Wu. 2024. Is DPO superior to PPO for LLM alignment? A comprehensive study. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

Feng Yao, Liyuan Liu, Dinghuai Zhang, Chengyu Dong, Jingbo Shang, and Jianfeng Gao. 2025. Your efficient rl framework secretly brings you off-policy rl training.

Chengzhi Yu, Yifan Xu, Yifan Chen, and Wenyi Zhang. 2025a. Optimizing lvlms with on-policy data for effective hallucination mitigation. *ArXiv preprint*, abs/2512.00706.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, and 16 others. 2025b. DAPO: An Open-Source LLM Reinforcement Learning System at Scale.

Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhu Chen. 2025. Acecoder: Acing coder rl via automated test-case synthesis. *arXiv preprint arXiv:2502.01718*.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023a. Self-edit: Fault-aware code editor for code generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 769–787. Association for Computational Linguistics.

Tianyi Zhang, Tao Yu, Tatsunori Hashimoto, Mike Lewis, Wen-Tau Yih, Daniel Fried, and Sida Wang. 2023b. Coder reviewer reranking for code generation. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 41832–41846. PMLR.

Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong Liu, Rui Men, An Yang, Jingren Zhou, and Junyang Lin. 2025. Group Sequence Policy Optimization.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Yefan Zhou, Austin Xu, Yilun Zhou, Janvijay Singh, Jiang Gui, and Shafiq Joty. 2025. Variation in verification: Understanding verification dynamics in large language models. *CoRR*, abs/2509.17995.

Xinyu Zhu, Mengzhou Xia, Zhepei Wei, Wei-Lin Chen, Danqi Chen, and Yu Meng. 2025. The surprising effectiveness of negative reinforcement in llm reasoning. *ArXiv preprint*, abs/2506.01347.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, and 1 others. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *ArXiv preprint*, abs/2406.15877.

Contents

A Dataset Statistics

Name	#Instances	#Questions	Avg. Sim.
Train	50000	1247	0.896
Heldout	3000	456	0.893
Strong	3000	1051	0.898
Hard	3000	137	0.931
Adv	18000	456	0.882

Table 6: **Dataset statistics.**

We present the detailed dataset statistics in Table 6. We represent each code using `Qwen3-Embedding-8B` because it achieves state-of-the-art results on MTEB (Muennighoff et al., 2023; Enevoldsen et al., 2025). The codes in `Aletheia-Hard` stand out from other datasets, having a much higher average similarity, which is anticipated because the incorrect codes also pass $70 - 90\%$ test cases. This explains why the scores for all models in the easy-to-hard setting are significantly lower.

B Additional Experiment Details

We use the implementation provided by Xiong et al. (2025) for RAFT and the `trl` library[5] for GRPO and DPO-Think. All training runs are conducted on a cluster of 8 NVIDIA H200 GPUs. To optimize memory usage, we employ Deepspeed ZeRO Stage-2 (Rasley et al., 2020) to shard activations and optimizer states across devices, and Flash-Attention 3 (Shah et al., 2024) to accelerate training. In all our training runs, we use the AdamW optimizer (Loshchilov and Hutter, 2019) with default parameters and a constant learning rate scheduler with 5% warmup steps, and train with an effective batch size of 64 for exactly 781 gradient steps to ensure a fair comparison.

Our GRPO implementation deviates significantly from the original (DeepSeek-AI, 2025) to incorporate future refinements to the recipe. We use the DAPO loss (Yu et al., 2025b) and Truncated Importance Sampling (Yao et al., 2025) with the truncation threshold set to 2.0. Although recent works have chosen to eliminate the KL coefficient, we set it to $\beta =$ 1e-3 because our base models are already fine-tuned to generate long reasoning traces. We synchronize the reference model every 100 steps (Gorbatovski et al., 2024; Liu et al., 2025a). We use a learning rate of 1e-6 and normalize by the standard deviation within each group. We note that while Liu et al. (2025c) suggests batch-level normalization for base models, their results indicate poor performance for aligned models, such as those used in this study. We generate a batch of 64 prompts and perform a single gradient update per batch, with $\epsilon_{\text{low}} = 0.2$ and $\epsilon_{\text{high}} = 0.28$. To encourage the model staying within budget, we use a soft overlong punishment reward (Yu et al., 2025b).

For batch-online GRPO, we use a generation batch of 256 prompts, performing 4 gradient updates per batch with $\epsilon_{\text{low}} =$ 3e-4, $\epsilon_{\text{high}} =$ 4e-4 and sequence-level importance sampling (Zheng et al., 2025). All other details are the same as the online GRPO variant.

Following Lambert et al. (2024), we train DPO with a learning rate of 5e-7, KL penalty $\beta = 0.1$, and an effective training batch size of 64. To reduce memory overhead, we precompute log-probabilities, eliminating the need to load the reference model during training. To train a DPO model, we also need an offline dataset of preferred and dispreferred generations. To this end, we sample

[5] https://github.com/huggingface/trl

100 outputs for each prompt in `Aletheia-Train` using `Deepseek-R1-Distill-Qwen-[1.5-32]B`, and score them with our verifiable reward function. For prompts with no correct answers, we use `Deepseek-R1`.

While prior work finds the quality of chosen responses to be more important (Pan et al., 2025b), we hypothesize that the reverse is true in a verifiable setting, where the quality of the "chosen" sample is fixed (correct), but the rejected quality can vary. Moreover, DPO is known to be sensitive to OOD shifts (Xu et al., 2024). Thus, we distribute the incorrect responses evenly between those generated by the 1.5–14B models. This also ensures that the negative samples for DPO come from generations similar to on-policy sampling. Our hypothesis is validated by the strong performance of our DPO models, even rivalling the fully online GRPO at larger sizes.

RAFT is trained with a learning rate of 2e-6 and an effective batch size of 64. Consistent with Dong et al. (2023); Xiong et al. (2025), no KL penalty is applied. In preliminary runs, we found that fine-tuning on the entire batch of correct responses leads to overfitting, especially in large models that generate a high proportion of correct responses. We mitigate this effect by fine-tuning on a maximum of 5 correct responses per group.

C Alternate Reward Formulations

Shaping the reward during RL training is a crucial decision, and numerous proposals for optimal reward functions have been made in prior work. We experiment with four reward formulations at 7B model scale and pick the best-performing one for our final training runs. The rewards used are as follows:

- **Pairwise Exact Match (PairEM)**. The simplest formulation - Given two candidates, we prompt the verifier to indicate its preference with a single token (A or B) within `boxed{}`.
- **Pairwise Scores (PairSc)**. This reward is taken from the JudgeLRM paper (Gandhi et al., 2025). The verifier outputs scores on a scale of $0-10$ for both candidate codes, and the reward is shaped based on accuracy, confidence, and format.
- **Listwise Exact Match (ListEM)**. A modified version of PairEM with between two and five candidates
- **Listwise Scores (ListSc)**. The verifier outputs scores out of 10 for each candidate, similar to PairSc. If the correct code is assigned the highest score, we assign a reward of $+1$, and award a bonus of $+1$ if this score is 10

Both listwise rewards are loosely based on DeepSeek-GRM (Liu et al., 2025d), adopted to our setting. For PairSc and ListSc, we use the pass rate of both codes as an indication of their quality. Since one of the codes is always correct, one of the scores outputted by the model should always be 10. We train these models using GRPO in the same manner as described in the main paper, and present the results in Table 7. We find that relatively simple ListEM works best, followed by PairSc.

Reward	Accuracy
PairEM	77.19
PairSc	78.24
ListEM	80.02
ListSc	77.36

Table 7: **Accuracy for alternate reward formulations**. All results are from training the 7B model for an equal number of gradient updates using GRPO. For a fair comparison, we evaluate on pairs of codes, which explains the higher absolute values as compared to Section 3.

D Modifications for `Aletheia-Adv`

Name	7B	14B	32B	Avg.
Positive Biases				
Authority Bias	**0.56**	**0.67**	**0.67**	**0.64**
Egocentric bias	0.52	0.49	0.54	0.52
External Reference	**0.58**	**0.78**	**0.85**	**0.73**
Bandwagon Effect	0.51	0.55	0.55	0.54
Illusory Complexity	0.40	0.44	0.49	0.44
Self-declared correctness	**0.64**	**0.77**	**0.75**	**0.72**
Negative Biases				
Minification	0.50	0.52	0.50	0.51
Misleading Comments	**0.53**	**0.76**	**0.82**	**0.71**
Renaming Identifiers	0.54	0.60	0.54	0.56
Reverse Authority Bias	**0.53**	**0.71**	**0.65**	**0.63**
Reverse Bandwagon Effect	0.44	0.60	0.56	0.53
Self-declared incorrectness	**0.60**	**0.81**	**0.86**	**0.76**

Table 8: **Modifications considered to construct `Aletheia-Adv`.** We report the Mean Influence Rate (MIR) for the 7–32B models, along with the average. Positive modifications are applied to the incorrect code, whereas negative ones are applied to the correct one. The top six modifications are highlighted.

For the creation of `Aletheia-Adv`, we experiment with several biasing factors based on prior work (Moon et al., 2025; Lam et al., 2025; Hwang et al., 2025; Bharadwaj et al., 2025; Wang et al., 2025a) as follows:

- **(Reverse) Authority Bias.** Prepends a comment that the code was written by an experienced (junior) developer.

- **(Reverse) Bandwagon Effect.** Indicates that a majority (minority) of developers prefer the ensuing code.
- **Egocentric Bias.** Indicates that an incorrect code was written by the evaluator.
- **External Reference.** Claims to be the reference solution on the competition's website.
- **Illusory complexity.** Add garbage or unreachable code to the existing code snippet, which may elicit length biases in the evaluator (Zheng et al., 2023).
- **Minification.** We use a rule-based minifier for C++ and Java, and `python-minifier`[6] for Python codes.
- **Misleading comments.** Adds misleading comments indicating the code makes an error.
- **Renaming identifiers.** We use the obfuscator provided by Paul et al. (2025) to obscure all variable, class, and function names.
- **Self-Declared (In)correctness.** Simply states that the code is (in)correct.

To analyze the vulnerability of the base models to these factors, we prompt `Deepseek-R1-Distill-Qwen2.5` $7-32$B on the original and perturbed versions of the same prompt, and measure how often the evaluator switches its answer.

We conduct evaluations in a pairwise setting. To isolate the effects of the biasing factors from position bias (Zheng et al., 2023), we report the Bias Influence Ratio (BIR), as the ratio between the number of times the LLM responds with the incorrect answer to the total number of switches. BIR close to 0 indicates an unbiased verifier, while easily biased LLMs would have BIRs close to 1. A BIR close to 0.5 indicates random switching, probably caused by position bias in the LLM. The results are as shown in Table 8.

Overall, we verify that LRMs are more robust to common biases that are prevalent in LLMs, as observed in prior work (Wang et al., 2025a). However, they are not completely unbiased, and we select the top six most misleading modifications to further use in our analysis of adversarial robustness in the main paper (Section 3).

[6] `https://github.com/dflook/python-minifier`

E Prompt Templates

Python code generation prompt

You are an expert Python programmer. You will be given a question (problem specification) and will generate a correct Python program that matches the specification and passes all tests. Read the inputs from STDIN solve the problem and write the answer to STDOUT (do not directly test on the sample inputs). Enclose your code within a Python markdown block. Ensure that when the python program runs, it reads the inputs, runs the algorithm and writes output to STDOUT.

C++ code generation prompt

You are an expert C++ programmer. You will be given a question (problem specification) and will generate a correct C++ program with a main function that matches the specification and passes all tests. Read the inputs from STDIN solve the problem and write the answer to STDOUT (do not directly test on the sample inputs). Enclose your code within a C++ markdown block. Ensure that when the C++ program runs, it reads the inputs, runs the algorithm and writes output to STDOUT.

Java code generation prompt

You are an expert Java programmer. You will be given a question (problem specification) and will generate a correct Java program with with a public class named Main that matches the specification and passes all tests. Your class should include a public static void main(String[] args) method. Read the inputs from System.in solve the problem and write the answer to System.out (do not directly test on the sample inputs). Enclose your code within a Java markdown block. Ensure that when the Java program runs, it reads the inputs, runs the algorithm and writes output to System.out.

Default training prompt

You are an expert judge of coding problems. Given a coding problem and multiple candidate solutions, your task is to evaluate the correctness of each solution based on the problem description. Your evaluation should solely be based on the functional correctness of the codes. It is guaranteed that one and only one of the candidates is completely correct. Here is the coding question followed by the candidate solutions:
[QUESTION]
{question}
[QUESTION]

[CANDIDATE_A]
{code_A}
[CANDIDATE_A]

[CANDIDATE_B] ...
You are to indicate your choice of candidate only by responding with one of the following options: {valid_options}. Enclose your final answer in the format `boxed{X}`, where X is your chosen option among the candidates. Do not provide any explanations or additional text. Your response should be exactly one of the options enclosed within
`boxed{}`, without any extra characters or spaces. Anything else will be considered invalid.

`GRPO-Instruct` training prompt

You are an expert judge of coding problems. Given a coding problem and multiple candidate solutions, your task is to evaluate the correctness of each solution based on the problem description. Your evaluation should solely be based on the functional correctness of the codes. It is guaranteed that one and only one of the candidates is completely correct. You are to indicate your choice of candidate only by responding with one of the following options: {valid_options}. Your response should be in the following format:
Analysis: <Your step-by-step reasoning here>
Final Answer:
`boxed{X}`, where X is your chosen option among the candidates.

Here is the coding question followed by the candidate solutions:
[QUESTION]
{question}
[QUESTION]

[CANDIDATE_A]
{code_A}
[CANDIDATE_A]

[CANDIDATE_B] ...
Your response should be exactly in the specified format, without any extra characters or spaces. Anything else will be considered invalid.

ListSc and PairSc training prompt

You are an expert judge of coding problems. Given a coding problem and two candidate solutions, your task is to evaluate the correctness of each solution based on the problem description. Your evaluation should solely be based on the functional correctness of the codes. It is guaranteed that one and only one of the candidates is completely correct. Here is the coding question followed by the candidate solutions:
[QUESTION]
{question}
[QUESTION]

[CANDIDATE_A]
{code_A}
[CANDIDATE_A]

[CANDIDATE_B] ...
You are to assign a score between 0 and 10 to EACH candidate, with 10 indicating a perfect solution that passes all test cases, 5 indicating a solution that would pass some test cases but not all, and 0 indicating a solution that fails all test cases. Output your final answer in the format `boxed{[<score_candidate_A>,<score_candid-ate_B>, <score_candidate_C>, ...]}` depending on the number of candidates. Do not provide any explanations or additional text. Your response should be a list of numbers between 0 and 10, enclosed within `boxed{}`, without any extra characters or spaces. Anything else will be considered invalid.