# Aletheia: What Makes RLVR For Code Verifiers Tick?

Anonymous ACL submission

## Abstract

Multi-domain thinking verifiers trained via Reinforcement Learning with Verifiable Rewards (RLVR) are a cornerstone of modern post-training. However, their adoption in code generation has lagged behind execution feedback due to the prohibitive costs of the full RLVR pipeline. In this work, we ablate three primary drivers of RLVR performance and cost: intermediate thinking traces, learning from negative samples, and on-policy training. We introduce **Aletheia**, a controlled, execution-grounded testbed to facilitate a contamination-free analysis of code verifiers across disparate model sizes and covariate shifts. Our analysis reveals that the optimal training recipe is scale-dependent: on-policy learning is the primary performance driver for small verifiers, whereas thinking traces become the most vital factor for larger sizes. Furthermore, we show that negative samples stabilize training at large sizes, and scaling inference-time compute cannot compensate for any core RLVR component. These findings provide a compute-optimal roadmap for practitioners, offering concrete strategies to simplify verifier training based on model size. Consequently, our work establishes a foundation for scalable supervision, enabling efficiently trained code verifiers to reliably supervise much larger code generation policies.

## 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).

Although automatic test-case generation is a common solution (Li et al., 2022; Liu et al., 2023b; Li et al., 2023a), they often struggle with test coverage and the inherent difficulty of specifying assertions for open-ended tasks. Alternate methods like self-contained environment creation (Jain et al., 2025; Xie et al., 2024) and world modelling (team et al., 2025) can be challenging for compiled languages without mature package managers.

In this work, we revisit surrogate code-execution verifiers (Ni et al., 2023; Li et al., 2025; Zeng et al., 2025; Shi et al., 2022; Zhang et al., 2023b): models trained to score code snippets based on execution outcomes without actually executing them. These verifiers can be integrated into post-training pipelines for code generation, obviating environment setup and code execution overhead. Additionally, these verifiers can leverage the code understanding and generalization capabilities of LLMs, and provide a granular feedback signal for long-horizon tasks where sparse episodic rewards may fail to guide convergence (Cui et al., 2025a).

Recently, thinking-enabled generative 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). Often, such verifiers are trained with RLVR (Chen et al., 2025d; Huang et al., 2025), boosting interpretability (Gunjal et al., 2025) and mitigating reward hacking (Chen et al., 2025b). Conversely, such advances have made fewer inroads in code generation (Ni et al., 2024; Shum et al., 2025), and the few existing code verifiers are encoder-only regression models (Zhu et al., 2026; Zhang et al., 2023b; Shi et al., 2022). This lack of adoption can be attributed to the prohibitive costs associated with the RLVR recipe, due to the complex orchestration of rollout, behavior, and reference policies. This raises a critical question: **Does robust code verification require the full RLVR suite, or can cheaper alternatives suffice?**

Core RLVR components have been extensively studied across various generative scenarios (Tajwar et al., 2024; Lanchantin et al., 2025; Zhu et al.,

We open source our code and datasets at: https://anonymous.4open.science/r/arr2026-aletheia/

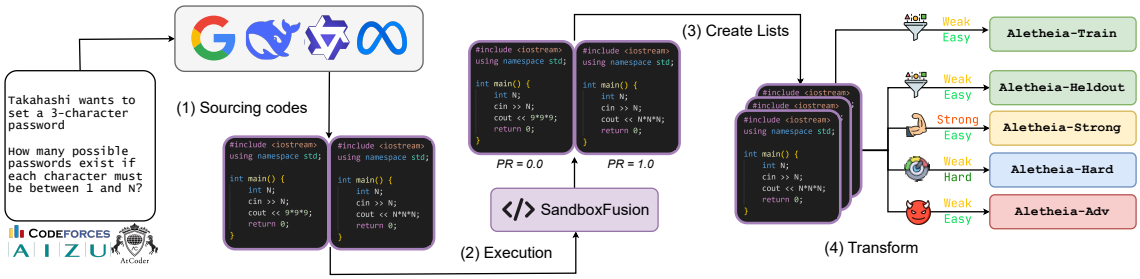


Figure 1: **The dataset creation pipeline.** We follow a four-step process: (1) sourcing competition-level programming questions from `CodeContests`$^+$ (Wang et al., 2025c) and sampling solutions from open-source LLMs; (2) executing these solutions via SandboxFusion to calculate pass rates (PRs); (3) constructing sets of $2-5$ candidate codes where exactly one is fully correct (PR $=$ 1.0); and (4) partitioning the resulting data into training and evaluation sets. Refer to Section 2.1 for a detailed description.

2025). However, a systematic analysis for verifiers is notably absent: a significant gap because generative performance in LLMs does not directly translate to verification capability (Li et al., 2023b). Although small regression models can approximate code execution outcomes (Akhauri et al., 2025), achieving robustness against adversarial or semantic-preserving transformations remains a challenge even for the largest off-the-shelf models (Haroon et al., 2025; Lyu et al., 2025).

In this work, we present a controlled analysis of the core components of RLVR by creating **`Aletheia`**: a contamination-free testbed that isolates algorithmic robustness to out-of-distribution (OOD) scenarios from data exposure, providing a rigorous benchmark for verifier performance. Across three model sizes ($1.5-14$B) and four evaluation axes, we ablate three primary drivers of performance and cost: generating long intermediate traces (**Thinking**), learning from both positive and negative samples (**Negatives**), and learning from data generated by an updated policy (**Online**).

Our analysis reveals that the optimal training configuration is scale-dependent. While on-policy learning is critical for small verifiers, its relative advantage diminishes as model size increases. Conversely, thinking traces offer limited benefits at smaller scales but become imperative for 14B models. Meanwhile, negative samples provide a consistent performance boost for all sizes and play a critical role in stabilizing training at larger scales. Additionally, the modest gains from scaling inference compute cannot compensate for any of the core components we analyze. To summarize, our contributions are as follows.

- We introduce **`Aletheia`**, an execution-grounded and cost-effective evaluation testbed. It enables contamination-free evaluation across three covariate shifts: stronger generators, harder comparisons, and adversarial responses (Section 2).
- We conduct an extensive ablation study across model scales ($1.5-14$B) to isolate three core RLVR components: thinking traces, on-policy learning, and negative samples. (Section 3).
- We identify several opportunities to simplify verifier training and provide concrete takeaways for practitioners (Section 4).

## 2 Experimental Setup

### 2.1 Data Curation and Testbed Creation

While direct deployment in a code-generation RLVR pipeline remains the gold standard for evaluating code verifiers, the computational cost is prohibitive for extensive ablations. Beyond the baseline requirement of doubling training resources to account for both $N$ verifiers and $N$ subsequent policy models, the inference latency of code verification is a severe bottleneck. Specifically, the reward computation for training generators would require $\approx$ 300ms per sample (Zhu et al., 2026), whereas exact-match rewards for training verifiers incur negligible cost ($<$ 1ms), a $\approx 300\times$ cost.

A common alternative is to use proxy evaluations that closely resemble real-world training scenarios (Frick et al., 2025; Lambert et al., 2025). Crucially, these evaluations must be free of data contamination to truly reflect a verifier's performance in an RLVR pipeline. Thus, we create `Aletheia`, a controlled and execution-grounded testbed to reliably evaluate our trained verifiers. It enforces a strict partition: training on data generated by weak models, easy comparisons, and benign prompts; but evaluating on data from stronger models, harder comparisons, or adversarial prompts. This setup mirrors prior work on scalable supervision (Burns et al., 2024; Hase et al., 2024; Sun et al., 2024), and ensures evaluation performance reflects algorithmic robustness to covariate shifts rather than data exposure (see Appendix F.2 for more details).

We define the verification task as follows: given a coding problem $x$ and a set of $N$ candidate solutions $Y = \{y_n\}_{n=1}^{N}$, the verifier must select the index $i$ of the correct solution $y_i \in Y$. We use competitive programming problems from `CodeContests`$^{+}$ (Wang et al., 2025c). They contain $\approx 25$ synthetic test cases per instance, which we filter to have true positive rates and true negative rates $> 0.9$ against a pre-evaluated user-submitted solution pool. We also discard samples with fewer than 5 test cases and a time limit exceeding 3 seconds. This gives us a dataset of 4903 programming questions with high-quality input-output test cases. We generate 50 completions for each problem in Python, C++, and Java at a high sampling temperature of 1.0 using a mix of `Weak` and `Strong` LLMs (refer to Table 6 in Appendix A).

The resulting code snippets are executed using `SandboxFusion`[1] to calculate their pass rates (PRs); the percentage of test cases passed for each code. We construct lists with $2-5$ solution codes, wherein each list contains codes with distinct PRs and exactly one with PR $= 1.0$. 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; `Easy` 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 harder to distinguish from the correct code.

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

Table 1: **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, evenly distributed by list length and programming language:

- **`Aletheia-Heldout`.** `Easy` comparisons by `Weak` models—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. Verifier performance on this dataset indicates their easy-to-hard generalization abilities (Hase et al., 2024; Sun et al., 2024).
- **`Aletheia-Adv`.** To evaluate the adversarial robustness of our verifiers. We apply three positive and negative modifications to the incorrect and correct codes in `Aletheia-Heldout` respectively (see Appendix D),[2] based on prior work on biases in LLM judges (LAM et al., 2025; Hwang et al., 2025; Moon et al., 2025).

All OOD evaluations introduce a unidirectional shift from the training data. We prevent data contamination by ensuring zero overlap between training and evaluation coding problems. The prompts used in our work are listed in Appendix G.

### 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 in Appendix B and experiment with alternate rewards in Appendix C.

## 3 Research Questions and Results

Our analysis employs a series of controlled experiments to isolate the contributions of Thinking, Negatives, and Online features. We use GRPO (Shao et al., 2024) as a baseline that includes all three components, and compare it with algorithms that lack the single component being studied (Table 1). This design choice eliminates confounding factors, ensuring observed performance reflects each component's individual contribution. We study the effect of removing two components in Appendix F.1.

### 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.`

[1] `bytedance/SandboxFusion`

[2] 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 | $38.78 \pm 0.85$ | $40.51 \pm 0.85$ | $31.22 \pm 0.81$ | $30.99 \pm 0.33$ | $35.41 \pm 0.71$ |
| GRPO-Think | | 4096 | 1.208 | $42.73 \pm 0.64$ | $40.70 \pm 0.61$ | $36.32 \pm 0.57$ | $31.15 \pm 0.25$ | $37.76 \pm 0.52$ |
| GRPO-Think | | 8192 | 2.491 | $46.82 \pm 0.62$ | $43.65 \pm 0.59$ | $41.61 \pm 0.61$ | $38.09 \pm 0.25$ | $42.55 \pm 0.52$ |
| GRPO-Think | | 16384 | 7.806 | $49.58 \pm 0.65$ | $46.09 \pm 0.62$ | $40.74 \pm 0.64$ | $40.97 \pm 0.26$ | $44.38 \pm 0.54$ |
| GRPO-Instruct | 7B | 4096 | 2.069 | $57.74 \pm 0.88$ | $51.80 \pm 0.89$ | $38.59 \pm 0.87$ | $52.20 \pm 0.37$ | $50.07 \pm 0.75$ |
| GRPO-Think | | 4096 | 3.561 | $59.54 \pm 0.71$ | $55.00 \pm 0.70$ | $46.73 \pm 0.74$ | $44.04 \pm 0.30$ | $51.32 \pm 0.61$ |
| GRPO-Think | | 8192 | 7.179 | $65.03 \pm 0.57$ | $56.96 \pm 0.58$ | $53.16 \pm 0.66$ | $52.03 \pm 0.26$ | $56.76 \pm 0.52$ |
| GRPO-Think | | 16384 | 15.101 | $74.81 \pm 0.57$ | $67.28 \pm 0.60$ | $53.11 \pm 0.69$ | $65.04 \pm 0.26$ | $65.05 \pm 0.53$ |
| GRPO-Instruct | 14B | 4096 | 9.463 | $63.45 \pm 0.82$ | $55.11 \pm 0.84$ | $44.15 \pm 0.84$ | $54.24 \pm 0.35$ | $54.26 \pm 0.71$ |
| GRPO-Think | | 4096 | 8.558 | $73.23 \pm 0.67$ | $64.95 \pm 0.71$ | $54.56 \pm 0.77$ | $58.09 \pm 0.31$ | $62.69 \pm 0.61$ |
| GRPO-Think | | 8192 | 14.900 | $78.37 \pm 0.60$ | $69.87 \pm 0.65$ | $61.74 \pm 0.73$ | $65.71 \pm 0.29$ | $68.91 \pm 0.57$ |
| GRPO-Think | | 16384 | 36.992 | $88.02 \pm 0.45$ | $83.65 \pm 0.49$ | $66.84 \pm 0.70$ | $83.67 \pm 0.21$ | $80.54 \pm 0.46$ |

Table 2: **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; Aljohani et al., 2025), 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). Thus, *long* intermediate chains may 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 evaluate the scaling behavior of inference-time generations $K$ and the training-time reasoning budget $B_{tr}$ (controlled by `trl.GRPOTrainer.max_completion_length`). We train four models, GRPO-Instruct with $B_{tr} = 4096$ and three GRPO-Think variants with $B_{tr} \in \{4096, 8192, 16384\}$. We evaluate our models using self-consistency (Wang et al., 2023), and summarize our results in Table 2. Costs are calculated assuming $10.6 per H200-hour.[3]

**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 value of verifiers in scalable supervision (Burns et al., 2024).

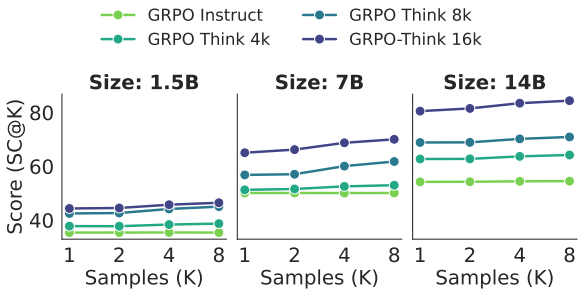


Figure 2: **Inference-time scaling for our thinking ablation.** Chain-of-Thought models do not benefit from additional compute, whereas thinking models see a small upward trend across all model sizes.

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 $7-14$B 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 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

[3] jarvislabs.ai/h200-price

that reasoning models are 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 aligns with prior work showing that reasoning traces enable 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, sampling a batch of responses and performing multiple gradient updates on mini-batches of generated data – a commonly-used compromise to speed up training (Zheng et al., 2025). We detail the DPO dataset creation in Appendix B and present the results for this ablation in Table 3.

**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 incurs a one-time cost, unlike 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.

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.

Batch-online GRPO fails to match GRPO performance, even underperforming DPO-Think in some OOD evaluation settings at larger sizes, and fails to provide a significant cost reduction compared to fully-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 (see Appendix B).

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-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.

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

Table 3: **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.

### 3.3 RQ3: Do negatives benefit code 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.

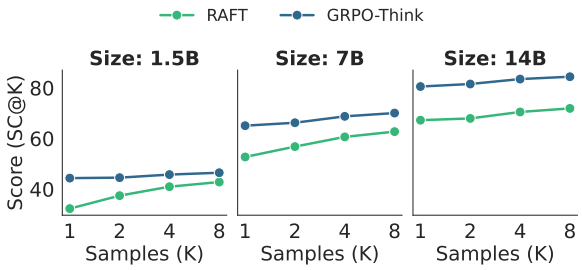


Figure 4: **Inference-time scaling plots for ablating negative samples.** The gap between RAFT and GRPO reduces with scaling compute, but persists.

**Setup.** We compare GRPO to a variant of RAFT (Dong et al., 2023), modified to use verifiable rewards (refer to Appendix E for a detailed description). RAFT works similarly to GRPO, sampling and scoring $N$ generations from a frequently updated policy, but simply trains on the correct responses using the next token prediction objective. We summarize our results in Table 4.

**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 compensate for negative samples.

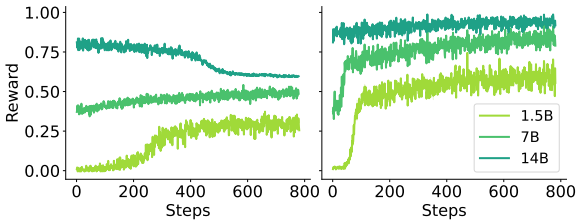


Figure 5: **Reward curves over training for RAFT and GRPO, respectively.** Training without negative samples becomes increasingly unstable with scale.

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.

## 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

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

Table 4: **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.

alternatives, we now translate these findings into actionable insights. This section provides actionable insights for practitioners optimizing verifier training during post-training.

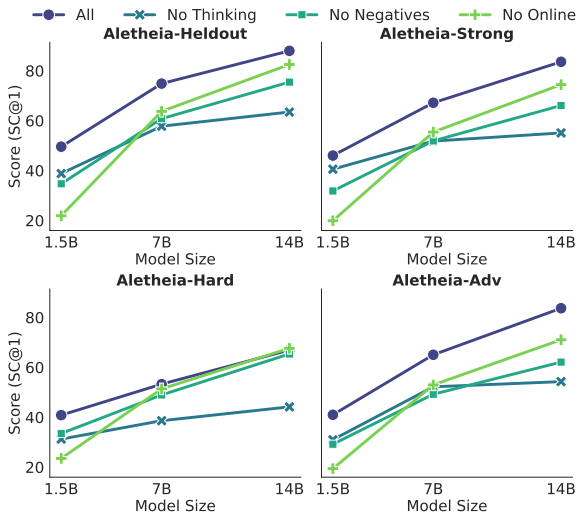


Figure 6: **Ablation of RLVR components across model sizes.** While removing any axis reduces accuracy, the bottleneck shifts with scale. Online learning is crucial for small models (1.5B), whereas Thinking becomes the dominant factor at larger scales (14B). Easy-to-Hard generalization is poor for all verifiers.

**RLVR is the best method to train verifiers.** Figure 6 shows that GRPO consistently matches or outperforms all ablated algorithms. Thus, we conclude that three components of RLVR training 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 $> 80\%$ accuracy, and further scaling would likely yield diminishing returns for the cost overheads.

**Thinking is critical for Easy-to-Hard generalization.** In the Aletheia-Hard setting, incorrect candidate programs pass 70–90% of test cases, a significant increase from the $< 50\%$ baseline in our training data. This higher pass rate induces greater lexical and functional similarity between candidates (Table 5 in Appendix A), increasing the complexity of the verification task. Consequently, all algorithms exhibit a marked performance degradation. Thus, thinking-enabled algorithms like DPO-Think and RAFT might be suitable alternatives to GRPO, particularly in the late-stage training regime where entropy collapse leads to highly homogeneous model outputs (Cui et al., 2025b).

**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, the gap widens drastically for the 14B model, with GRPO-Instruct not improving from its 7B variant. This indicates that vanilla Chain-of-Thought prompting cannot fully utilize the model's capacity beyond a certain point (around $7-8$B parameters), and that 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. Moreover, negative samples play a crucial role in stabilizing training at larger sizes, as shown in Figure 5. This confirms 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 code verifier performance.** Generative verifiers have a unique advantage over scalar reward models via test-time scaling. However, our experiments on code verifiers demonstrate that such scaling offers nominal 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 SC@1; while increased compute reduces the performance gap relative to GRPO-trained baselines, it does not fully bridge it. We provide additional evidence for this behaviour in Appendix F.3

## 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 components 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 et al., 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 (Zhu et al., 2026; Ni et al., 2024) or indirectly (team 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 show that RLVR enables the training of robust code verifiers that can scalably supervise much larger policy models, and using a subset of its core components can yield similar results depending on model scale and evaluation scenarios.

**Prior analyses of RLVR in LLM training.** Modern RL training is notoriously compute-intensive and inefficient (Noukhovitch et al., 2025; Piché et al., 2025), prompting a surge into simplifying the RL recipe by omitting core components like long thinking traces (Arora and Zanette, 2025; Sui et al., 2025), negative samples (Dong et al., 2023; Gulcehre et al., 2023; Tan et al., 2025), and on-policy learning (Rafailov et al., 2023; Wang et al., 2025b). However, the contribution of these components to RLVR's success is unclear, as described in Sections 3.1 to 3.3. Moreover, they are often treated as isolated rather than synergistic contributors to RLVR's success. A notable exception is Tajwar et al. (2024), who found that on-policy learning and negative samples are complementary and especially important when high-reward responses are less likely under the policy distribution. Additionally, most RLVR studies focus on mathematical reasoning. It is unclear whether these findings transfer to more brittle domains like code verification, where even frontier models frequently fail (Haroon et al., 2025; Lyu et al., 2025). In this work, we deconstruct the training dynamics of code verifiers to reveal that the importance of specific components is not static, but varies with scale. We provide a roadmap for efficiently training code verifiers to supervise future generations of code models.

## 6 Conclusion

In this work, we presented a systematic analysis of the three primary drivers of performance and cost in the RLVR pipeline for code verifiers: intermediate thinking traces, learning from negative samples, and on-policy training. To facilitate this study, we introduced **Aletheia**, a controlled, execution-grounded testbed designed for the contamination-free evaluation of code verifiers across disparate model sizes and covariate shifts. Our analysis reveals that these components are synergistic, and the optimal code verifier training recipe is scale-dependent: while on-policy learning is the primary performance driver for small verifiers, thinking traces become the most vital factor as model size increases. Furthermore, our results show that negative samples consistently boost performance and are crucial for stabilizing training at larger scales, and that scaling inference-time compute cannot compensate for the absence of core RLVR components. These findings establish a compute-optimal roadmap for practitioners, providing concrete strategies to simplify verifier training. More broadly, our work lays the foundation for code verifiers being a prominent fixture of the LLM post-training pipeline, allowing them to be trained efficiently and scalably supervise much larger models.

## Limitations

**Choice of optimization algorithms.** Although we holistically and experimentally verify our RL training recipe, which serves as our baseline for 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$^{EM}$ (Tan et al., 2025), Iterative DPO (Xu et al., 2023; Xiong et al., 2024), NSR (Zhu et al., 2025), etc.

**Analysis-focused training intent.** Our primary goal in this paper was to analyze the components of the RLVR recipe rather than training a state-of-the-art code verifier. `Aletheia-Train` contains `Easy` comparisons generated by `Weak` models, ensuring evaluation scores reflect algorithmic robustness to OOD shifts rather than simple data exposure. This segregation allows us to completely disentangle the training and evaluation data distributions and reliably predict the performance of our verifiers in downstream RLVR integrations. However, it is suboptimal for practitioners seeking to maximize verifier performance to follow our data curation strategy, as in-distribution performance would naturally exceed OOD performance. Thus, we suggest that practitioners train on a broader set of anticipated scenarios to improve the downstream robustness of the resulting verifier.

**Cross-domain generalizability.** Our work focuses on verifiers for competitive programming, an area that remains underexplored compared to standard reward modeling, as described in Section 1. Code contests offer a unique advantage for our research: they provide a high-precision mechanism for controlling *comparison difficulty* via granular test-case 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 meticulous curation of candidates during training.

Constructing a testbed for more complex code-related tasks, such as long-horizon software engineering or library-use problems, is an incredibly exciting horizon. However, creating an equivalent, controlled, and execution-grounded testbed such as ours for these domains is a significant challenge due to the absence of an objective, self-contained scoring rubric, making it a potential direction for future research. Our framework intentionally avoids code-specific heuristics or domain-dependent rewards to be broadly applicable; instead treating code execution purely as a high-fidelity ground truth for a simple binary reward signal. Future work can build on our methods to empirically verify our insights in other domains.

## 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 that our verifiers are susceptible to reward hacking – assigning high scores to responses that are incorrect or potentially harmful. We take steps to mitigate these risks by thoroughly documenting our entire workflow and making our datasets, codes, and models open-source under the `CC BY-NC-SA 4.0` License.

## References


Arash Ahmadian, Chris Cremer, Matthias Gallé, Marzieh Fadaee, Julia Kreutzer, Olivier Pietquin, Ahmet Üstün, and Sara Hooker. 2024. Back to basics: Revisiting REINFORCE-style optimization for learning from human feedback in LLMs. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12248–12267, Bangkok, Thailand. Association for Computational Linguistics.

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

Manar Aljohani, Jun Hou, Sindhura Kommu, and Xuan Wang. 2025. A comprehensive survey on the trustworthiness of large language models in healthcare. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 6720–6748, Suzhou, China. Association for Computational Linguistics.

Charles Arnal, Gaëtan Narozniak, Vivien Cabannes, Yunhao Tang, Julia Kempe, and Remi Munos. 2025. Asymmetric REINFORCE for off-policy reinforcement learning: Balancing positive and negative rewards. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Daman Arora and Andrea Zanette. 2025. Training language models to reason efficiently. *CoRR*, abs/2502.04463.

Anirudh Bharadwaj, Chaitanya Malaviya, Nitish Joshi, and Mark Yatskar. 2025. Flattery, fluff, and fog:

Diagnosing and mitigating idiosyncratic biases in preference models. *CoRR*, abs/2506.05339.

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.

Howard Chen, Noam Razin, Karthik Narasimhan, and Danqi Chen. 2025a. Retaining by doing: The role of on-policy data in mitigating forgetting. *CoRR*, 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. *CoRR*, 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. *CoRR*, 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. *CoRR*, abs/2505.02387.

James Chua and Owain Evans. 2025. Are deepseek r1 and other reasoning models more faithful? *CoRR*, abs/2501.08156.

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. 2025a. Process reinforcement through implicit rewards. *CoRR*, abs/2502.01456.

Ganqu Cui, Yuchen Zhang, Jiacheng Chen, Lifan Yuan, Zhi Wang, Yuxin Zuo, Haozhan Li, Yuchen Fan, Huayu Chen, Weize Chen, Zhiyuan Liu, Hao Peng, Lei Bai, Wanli Ouyang, Yu Cheng, Bowen Zhou, and Ning Ding. 2025b. The Entropy Mechanism of Reinforcement Learning for Reasoning Language Models. *CoRR*, abs/2505.22617.

DeepSeek-AI. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *CoRR*, abs/2501.12948.

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. *CoRR*, abs/2501.12599.

Kenneth C. Enevoldsen, Isaac Chung, Imene Kerboua, Márton Kardos, Ashwin Mathur, David Stap, Jay Gala, Wissam Siblini, Dominik Krzeminski, Genta Indra Winata, Saba Sturua, Saiteja Utpala, Mathieu Ciancone, Marion Schaeffer, Diganta Misra, Shreeya Dhakal, Jonathan Rystrøm, Roman Solomatin, Ömer Veysel Çagatan, and 2 others. 2025. MMTEB: massive multilingual text embedding benchmark. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Evan Frick, Tianle Li, Connor Chen, Wei-Lin Chiang, Anastasios Nikolas Angelopoulos, Jiantao Jiao, Banghua Zhu, Joseph E. Gonzalez, and Ion Stoica. 2025. How to evaluate reward models for RLHF. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Kanishk Gandhi, Ayush K Chakravarthy, Anikait Singh, Nathan Lile, and Noah Goodman. 2025. Cognitive behaviors that enable self-improving reasoners, or, four habits of highly effective STaRs. In *Second Conference on Language Modeling*.

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.

2025. Learn your reference model for real good alignment. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Caglar Gulcehre, Tom Le Paine, Srivatsan Srinivasan, Ksenia Konyushkova, Lotte Weerts, Abhishek Sharma, Aditya Siddhant, Alex Ahern, Miaosen Wang, Chenjie Gu, Wolfgang Macherey, Arnaud Doucet, Orhan Firat, and Nando de Freitas. 2023. Reinforced Self-Training (ReST) for Language Modeling. *CoRR*, abs/2308.08998.

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. *CoRR*, abs/2505.14674.

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. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 7002–7024. Association for Computational Linguistics.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, and Boris Ginsburg. 2024. RULER: What's the real context size of your long-context language models? In *First Conference on Language Modeling*.

Jian Hu, Jason Klein Liu, Haotian Xu, and Wei Shen. 2025. Reinforce++: Stabilizing critic-free policy optimization with global advantage normalization. *CoRR*, abs/2501.03262.

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. *CoRR*, 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. *CoRR*, abs/2508.07805.

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, Tianjun Zhang, Liang Zheng, Koushik Sen, and Ion Stoica. 2025. R2e-gym: Procedural environment generation and hybrid verifiers for scaling open-weights SWE agents. In *Second Conference on Language Modeling*.

Subbarao Kambhampati, Kaya Stechly, Karthik Valmeekam, Lucas Paul Saldyt, Siddhant Bhambri, Vardhan Palod, Atharva Gundawar, Soumya Rani Samineni, Durgesh Kalwar, and Upasana Biswas. 2025. Stop anthropomorphizing intermediate tokens as reasoning/thinking traces! In *NeurIPS 2025 Workshop on Bridging Language, Agent, and World Models for Reasoning and Planning*.

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 Lyu. 2025. Codecrash: Exposing LLM fragility to misleading natural language in code reasoning. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

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. *CoRR*, abs/2411.15124.

Nathan Lambert, Valentina Pyatkin, Jacob Morrison, LJ Miranda, Bill Yuchen Lin, Khyathi Chandu, Nouha Dziri, Sachin Kumar, Tom Zick, Yejin Choi, Noah A. Smith, and Hannaneh Hajishirzi. 2025. RewardBench: Evaluating reward models for language modeling. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 1755–1797, Albuquerque, New Mexico. Association for Computational Linguistics.

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. *CoRR*, abs/2506.21495.

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, Vienna, Austria, July 27 - August 1, 2025*, pages 8169–8182. Association for Computational Linguistics.

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

Xiang Lisa Li, Vaishnavi Shrivastava, Siyan Li, Tatsunori Hashimoto, and Percy Liang. 2023b. Benchmarking and improving generator-validator consistency of language models. *CoRR*, abs/2310.01846.

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. *Transactions on Machine Learning Research*, 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 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

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. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

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. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 865–883, Miami, Florida, USA. Association for Computational Linguistics.

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. *CoRR*, abs/2508.08221.

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. *CoRR*, abs/2504.02495.

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, Zichen Liang, Qian Sun, and Jiaming Zhang. 2025. Surge: On the potential of large language models as general-purpose surrogate code executors. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 3268–3308, Suzhou, China. Association for Computational Linguistics.

Xueguang Ma, Qian Liu, Dongfu Jiang, Ge Zhang, Zejun MA, and Wenhu Chen. 2025. General-reasoner: Advancing LLM reasoning across all domains. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

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. *CoRR*, abs/2505.16222.

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 I. Wang, and Xi Victoria Lin. 2023. LEVER: learning to verify language-to-code generation with execution. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, 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, :, Aaron Blakeman, Aaron Grattafiori, Aarti Basant, Abhibha Gupta, Abhinav Khattar, Adi Renduchintala, Aditya Vavre, Akanksha Shukla, Akhiad Bercovich, Aleksander Ficek, Aleksandr Shaposhnikov, Alex Kondratenko, Alexander Bukharin, Alexandre Milesi, Ali Taghibakhshi, Alisa Liu, Amelia Barton, and 295 others. 2025. Nemotron 3 nano: Open, efficient mixture-of-experts hybrid mamba-transformer model for agentic reasoning. *CoRR*, abs/2512.20848.

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, Huaiyang Zhong, Guanting Chen, and Chonghuan Wang. 2025b. What matters in data for DPO? In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Indraneil Paul, Haoyi Yang, Goran Glavaš, Kristian Kersting, and Iryna Gurevych. 2025. Obscuracoder: Powering efficient code LM pre-training via obfuscation grounding. In *The Thirteenth International Conference on Learning Representations*.

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

Julian Aron Prenner and Romain Robbes. 2023. Runbugrun – an executable dataset for automated program repair. *CoRR*, abs/2304.01102.

Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir R. Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. Codenet: A large-scale AI for code dataset for learning a diversity of coding tasks. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

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*.

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. *CoRR*, abs/2402.03300.

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*, pages 3533–3546, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K. Reddy. 2023. Execution-based code generation using deep reinforcement learning. *Transactions on Machine Learning Research*, 2023.

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. *CoRR*, abs/2512.21919.

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, Vardhan Palod, Atharva Gundawar, and Subbarao Kambhampati. 2025. Beyond semantics: The unreasonable effectiveness of reasonless intermediate tokens. In *First Workshop on Foundations of Reasoning in Language Models*.

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. *Transactions on Machine Learning Research*.

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.

Wentao Tan, Qiong Cao, Yibing Zhan, Chao Xue, and Changxing Ding. 2025. Beyond human data: Aligning multimodal large language models by iterative self-evolution. In *AAAI-25, Sponsored by the Association for the Advancement of Artificial Intelligence, February 25 - March 4, 2025, Philadelphia, PA, USA*, pages 7202–7210. AAAI Press.

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. *CoRR*, abs/2405.08448.

FAIR CodeGen team, 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, and 32 others. 2025. CWM: an open-weights LLM for research on code generation with world models. *CoRR*, abs/2510.02387.

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. In *Second Conference on Language Modeling*.

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.

Yubo Wang, Xiang Yue, and Wenhu Chen. 2025b. Critique Fine-Tuning: Learning to Critique is More Effective than Learning to Imitate. *CoRR*, abs/2501.17703.

Zihan Wang, Siyao Liu, Yang Sun, Ming Ding, and Hongyan Li. 2025c. CodeContests+: High-quality test case generation for competitive programming. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 5576–5600, Suzhou, China. Association for Computational Linguistics.

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. *CoRR*, abs/2505.10320.

Fang Wu, Weihao Xuan, Ximing Lu, Mingjie Liu, Yi Dong, Zaid Harchaoui, and Yejin Choi. 2025. The invisible leash: Why rlvr may or may not escape its origin. *CoRR*, abs/2507.14843.

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. *CoRR*, abs/2504.11343.

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. *CoRR*, 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. *Feng Yao's Notion*.

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

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, YuYue, Weinan Dai, Tiantian Fan, Gaohong Liu, Juncai Liu, LingJun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, and 17 others. 2025b. DAPO: An open-source LLM reinforcement learning system at scale. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Yang Yue, Zhiqi Chen, Rui Lu, Andrew Zhao, Zhaokai Wang, Yang Yue, Shiji Song, and Gao Huang. 2025. Does Reinforcement Learning Really Incentivize Reasoning Capacity in LLMs Beyond the Base Model? *CoRR*, abs/2504.13837.

Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhu Chen. 2025. ACECODER: acing coder RL via automated test-case synthesis. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pages 12023–12040. Association for Computational Linguistics.

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)*, pages 769–787, Toronto, Canada. 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. *CoRR*, abs/2507.18071.

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.

Xiao Zhu, Xinyu Zhou, Boyu Zhu, Hanxu Hu, Mingzhe Du, Haotian Zhang, Huiming Wang, and Zhijiang Guo. 2026. Codescaler: Scaling code llm training and test-time inference via execution-free reward models. *CoRR*, abs/2602.17684.

Xinyu Zhu, Mengzhou Xia, Zhepei Wei, Wei-Lin Chen, Danqi Chen, and Yu Meng. 2025. The surprising effectiveness of negative reinforcement in LLM reasoning. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, James Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, and 2 others. 2025. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

# Contents


**1 Introduction** **1**

**2 Experimental Setup** **2**
- 2.1 Data Curation and Testbed Creation 2
- 2.2 Training details . . . . . 3

**3 Research Questions and Results** **3**
- 3.1 RQ1: Do verifiers need thinking traces? . . . . . 3
- 3.2 RQ2: Is on-policy learning essential for verifier training? . . . . . 5
- 3.3 RQ3: Do negatives benefit code verifiers? . . . . . 6

**4 Discussion and Takeaways** **6**

**5 Related Work** **8**

**6 Conclusion** **8**

**A Dataset Statistics** **16**

**B Additional Experiment Details** **16**

**C Alternate Reward Formulations** **17**

**D Modifications for Aletheia-Adv** **17**

**E Other RAFT variants** **18**

**F Supporting Results** **19**
- F.1 Supervised Fine-tuning . . . . . 19
- F.2 Training on a mixed dataset . . . . 19
- F.3 Inference-time scaling on RunBugRun . . . . . 20

**G Prompt Templates** **20**


# A Dataset Statistics

| Name | #Instances | #Questions | Avg. Sim. |
|---|---|---|---|
| Aletheia-Train | 50,000 | 1247 | 0.896 |
| Aletheia-Heldout | 3000 | 456 | 0.893 |
| Aletheia-Strong | 3000 | 1051 | 0.898 |
| Aletheia-Hard | 3000 | 137 | 0.931 |
| Aletheia-Adv | 18,000 | 456 | 0.882 |

Table 5: **Dataset statistics.** Average similarities are calculated by computing the cosine similarities between the embeddings of the codes in each list.

We present the detailed dataset statistics in Table 5, and the generators used in our datasets in Table 6. The strength of an LLM is determined based on its size and its standing on the BigCodeBench Leaderboard (Zhuo et al., 2025). 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.

| 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 6: **Generators used for our datasets.** ΔScore is the difference between the Pass@1 BigCodeBench-Instruct scores (Zhuo et al., 2025) of the Strong and Weak models. We use the -Instruct variant of all models in out experiments.

# B Additional Experiment Details

We use the implementation provided by Xiong et al. (2025) for RAFT and the trl library[4] 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 $\beta = 1\text{e-}3$ because our base models are already fine-tuned to generate long reasoning traces. We synchronize the reference model every 100 steps (Gorbatovski et al., 2025; 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 up-

[4] huggingface/trl

date 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 $\epsilon_{\text{low}} = 3\text{e-}4$, $\epsilon_{\text{high}} = 4\text{e-}4$ 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 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 $1.5 - 14$B 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 (See Appendix E for more details).

## 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. Moreover, we find that our claims about inference-time scaling trends hold even for alternative reward formulations, failing to yield significant gains in any setting.

| Reward | Avg. SC@1 | Avg. SC@2 | Avg. SC@4 | Avg. SC@8 |
|---|---|---|---|---|
| PairSC | $78.24 \pm 0.93$ | $78.13 \pm 0.93$ | $80.16 \pm 1.0$ | $82.22 \pm 1.06$ |
| PairEM | $77.19 \pm 0.90$ | $77.50 \pm 0.90$ | $78.93 \pm 0.95$ | $80.12 \pm 1.00$ |
| ListSC | $77.36 \pm 0.89$ | $77.32 \pm 0.89$ | $79.40 \pm 0.94$ | $80.82 \pm 0.98$ |
| ListEM | $80.02 \pm 0.92$ | $80.14 \pm 0.92$ | $81.50 \pm 0.94$ | $83.02 \pm 0.98$ |

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`

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.

| 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.

- **(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, eliciting length biases in the evaluator (Zheng et al., 2023).
- **Minification.** We use a rule-based minifier for C++ and Java, and `python-minifier`[5] 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` from 7B to 32B parameters 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 for analysis of adversarial robustness in the main text.

## E Other RAFT variants

While designing our Negatives ablation in Section 3.3, we experimented with several variants of the RAFT algorithm. A crucial requirement was that our RAFT algorithm be fully on-policy to isolate the effect of negative samples. The original RAFT (Dong et al., 2023), akin to common rejection-sampling algorithms, is batch-online: sampling $N$ responses each for a batch of prompts from the current model, scoring them using a reward model, and fine-tuning the current model on the $K$ highest scoring prompt-response pairs using a negative log-likelihood loss. Moreover, this algorithm was designed for RLHF-style reward models that output a continuous scalar score, unlike the binary reward signals prevalent in GRPO.

Our implementation closely follows Xiong et al. (2024), who adopt RAFT to the RLVR setting, with some modifications. Most importantly, we collect new data after a single gradient update (which slows down training, but ensures training remains fully on-policy). An unclear implementation detail from their work is whether they train on *all* the correct responses for a group or a subset of them. We experiment with both variants, as well as the RAFT++ algorithm proposed by Xiong et al. (2024), and present the results for the 14B model in Table 9.

| Algorithm | Avg `SC@1` | Description |
|---|---|---|
| RAFT | $51.20 \pm 0.53$ | RAFT adapted to verifiable rewards |
| **RAFT-max5** | $\mathbf{67.20 \pm 0.52}$ | RAFT trained on a maximum of 5 correct responses per group |
| RAFT++ | $60.41 \pm 0.47$ | Variant adding importance sampling and clipping to RAFT |

Table 9: **Experimental results for RAFT variants.** We present the average `SC@1` scores across the four `Aletheia-` evaluation datasets, along with their 95% confidence interval.

Clearly, fine-tuning on all correct responses for each group leads to overfitting on easy examples, yielding significantly more correct responses than on harder ones. Surprisingly, adding PPO-style

[5] `dflook/python-minifier`

importance sampling and clipping techniques does not stabilize training either, further emphasizing the role of negative samples. We use the best performing max-5 variant of RAFT for the results reported in the main text.

# F Supporting Results

## F.1 Supervised Fine-tuning

We train models using the Supervised Fine-Tuning (SFT) objective on the positively scored responses from `Aletheia-DPO` across the $1.5-14$B scale to further validate our analyses of Negatives and Online components. Notably, we omit this algorithm from the main text because it differs from GRPO along two axes: Negatives and Online. However, in this section, we compare it to the other "incomplete" algorithms: Think-DPO, BatchOnline (BO-) GRPO, and RAFT. We report results in Table 10.

| Algorithm | Size | Avg SC@1 | Thinking | Negatives | Online |
|---|---|---|---|---|---|
| SFT | 1.5B | $18.00 \pm 0.57$ | ✔ | ✘ | ✘ |
| Think-DPO | | $21.11 \pm 0.31$ | ✔ | ✔ | ✘ |
| BO-GRPO | | $38.99 \pm 0.46$ | ✔ | ✔ | ~ |
| RAFT | | $32.30 \pm 0.39$ | ✔ | ✘ | ✔ |
| GRPO-Think | | $44.38 \pm 0.54$ | ✔ | ✔ | ✔ |
| SFT | 7B | $41.11 \pm 0.73$ | ✔ | ✘ | ✘ |
| Think-DPO | | $55.88 \pm 0.50$ | ✔ | ✔ | ✘ |
| BO-GRPO | | $55.46 \pm 0.54$ | ✔ | ✔ | ~ |
| RAFT | | $52.72 \pm 0.51$ | ✔ | ✘ | ✔ |
| GRPO-Think | | $65.05 \pm 0.53$ | ✔ | ✔ | ✔ |
| SFT | 14B | $58.60 \pm 0.55$ | ✔ | ✘ | ✘ |
| Think-DPO | | $73.89 \pm 0.51$ | ✔ | ✔ | ✘ |
| BO-GRPO | | $75.29 \pm 0.50$ | ✔ | ✔ | ~ |
| RAFT | | $67.20 \pm 0.52$ | ✔ | ✘ | ✔ |
| GRPO-Think | | $80.54 \pm 0.46$ | ✔ | ✔ | ✔ |

Table 10: **Additional results on an SFT baseline for the 14B models.** We report `SC@1` scores and their 95% confidence interval. At larger scales, on-policy learning is the least critical factor, and negatives play a critical role in training stability for all sizes.

Removing two of the core components from GRPO significantly hampers performance across all model sizes. Our observations about the importance of on-policy learning decreasing with scale remain valid, as evidenced by the SFT–RAFT gap decreasing with scale (from 14.3% to 8.6%). Crucially, the utility of negative samples for stabilizing training is more pronounced in these ablations, as shown by comparing the SFT–RAFT gap to the SFT–(BO-GRPO) gap. Despite BO-GRPO being only partially on-policy, it outperforms the fully on-policy RAFT across all model scales due to the presence of negative samples. At small scales, using batch-online GRPO: a mixture of on-policy training and negatives, is the best alternative to the full GRPO algorithm. However, at medium–large scales, using a completely offline algorithm (DPO-Think) can already yield good results at much lower cost, as mentioned in Section 4.

| Algorithm | `Aletheia-Strong` | `Aletheia-Hard` | `Aletheia-Adv` |
|---|---|---|---|
| GRPO-Think | $41.46 \pm 0.58$ | $39.05 \pm 0.57$ | $34.09 \pm 0.24$ |
| GRPO-Instruct | $37.11 \pm 0.87$ | $29.68 \pm 0.82$ | $31.61 \pm 0.34$ |
| DPO-Think | $20.37 \pm 0.39$ | $24.24 \pm 0.43$ | $20.08 \pm 0.17$ |
| BO-GRPO | $40.11 \pm 0.54$ | $39.19 \pm 0.54$ | $33.42 \pm 0.23$ |
| RAFT | $32.99 \pm 0.49$ | $34.06 \pm 0.50$ | $29.67 \pm 0.21$ |

Table 11: **Results from training 1.5B verifiers on `Aletheia-Mixed`.** We report the SC@1 scores with their 95% confidence intervals. Training on a mixed dataset yields minor performance boosts over the `Aletheia` testbed in most cases, but sacrifices conclusions about OOD robustness.

## F.2 Training on a mixed dataset

A key advantage of the `Aletheia` testbed is the complete separation of training and evaluation data distributions, enabling an accurate estimation of the OOD robustness of our trained verifiers. However, this setting differs from the practical approach: training verifiers on a mix of all anticipated scenarios to achieve the best downstream performance. For completeness, we present the results from training a 1.5B model on a mixture of all our evaluation scenarios. Concretely, we utilized unused instances from the `Weak-Hard` and `Strong-Easy` buckets (i.e., instances that do not appear in the corresponding evaluation sets), and create a new adversarial dataset by perturbing `Aletheia-Train` using the six best modifications detailed in Appendix D. Crucially, we now apply the modifications at random rather than targeting the correct or incorrect code to avoid leaking any information about the ground truth and to train the verifier to be robust to such perturbations. The resulting `Aletheia-Mixed` contains all four data distributions in equal proportions. We create a corresponding DPO dataset following the same procedure as in Appendix B, and summarize our results in Table 11.

Training on a mixture of all tasks yields small performance gains for most algorithms, as compared to the results in Tables 2 to 4. More importantly, our conclusions from the main text at the 1.5B scale still hold. DPO has the worst performance at this scale, and introducing even a semi-on-policy update can recover most of the offline-online performance gap, making on-policy learning the most crucial component at this scale.

Crucially, training on a mixed dataset sacrifices the ability to analyze the OOD robustness of the studied algorithms. Since `Aletheia-Mixed` contains a mixture of all anticipated evaluation scenarios, all three evaluation axes are in-distribution, providing no signal on the robustness of these verifiers in a downstream RLVR pipeline. Thus, by

completely separating the training and evaluation data distributions, the `Aletheia` testbed provides a foundation for our controlled analysis, enabling us to stress-test verifiers in a proxy-evaluation setting without incurring prohibitive costs.

### F.3 Inference-time scaling on RunBugRun

To ensure the validity of our inference-time scaling observations across reward formulations, we further test our hypothesis on the RunBugRun dataset (Prenner and Robbes, 2023), which contains correct-bugged code pairs from CodeNet (Puri et al., 2021). We report our results in Table 12.

| Reward | RBR SC@1 | RBR SC@2 | RBR SC@4 | RBR SC@8 |
|---|---|---|---|---|
| PairSC | $67.43 \pm 1.12$ | $67.23 \pm 1.12$ | $69.48 \pm 1.21$ | $71.08 \pm 1.30$ |
| PairEM | $70.84 \pm 1.14$ | $70.83 \pm 1.13$ | $72.99 \pm 1.22$ | $73.61 \pm 1.30$ |
| ListSC | $71.01 \pm 1.21$ | $71.26 \pm 1.21$ | $73.12 \pm 1.29$ | $74.17 \pm 1.37$ |
| ListEM | $73.24 \pm 1.14$ | $73.63 \pm 1.13$ | $74.99 \pm 1.22$ | $75.61 \pm 1.30$ |

Table 12: **Inference scaling trends on the RunBugRun dataset.** GRPO-trained models see limited gains from inference compute scaling.

The modest gains from inference-time scaling persist across an external dataset, demonstrating that our observation is a consistent pattern rather than an artifact of our specific setup. Moreover, it is supported by prior literature on entropy collapse in LLMs, which finds that RLVR merely improves the sampling efficiency of language models and can potentially shrink the space of accessible reasoning paths (Yue et al., 2025; Wu et al., 2025).

## G 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_candidate_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.