BREAKING TRAINING BOTTLENECKS: EFFECTIVE AND STABLE REINFORCEMENT LEARNING FOR CODING MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Modern code generation models exhibit longer outputs, accelerated capability growth, and changed training dynamics, rendering traditional training methodologies, algorithms, and datasets ineffective for improving their performance. To address these training bottlenecks, we propose **MicroCoder-GRPO**, an improved Group Relative Policy Optimization approach with two original innovations and one inherited improvement: conditional truncation masking to improve long output potential while maintaining training stability, diversity-determined temperature selection to maintain and encourage output diversity, and removal of KL loss with high clipping ratios following DAPO to facilitate solution diversity. MicroCoder-GRPO achieves up to 17.6% relative improvement over strong baselines on LiveCodeBench v6, with more pronounced gains under extended context evaluation. Additionally, we release **MicroCoder-Dataset**, a more challenging training corpus that achieves 3x larger performance gains than mainstream datasets on LiveCodeBench v6 within 300 training steps, and **MicroCoder-Evaluator**, a robust framework that reduces disagreement with human labels from 10.8% to 1.4% and around 40% faster execution through parallel processing. Through comprehensive analysis across more than thirty controlled experiments, we reveal **training insights** across seven main aspects, demonstrating that properly trained models can achieve competitive performance with larger counterparts.

1 INTRODUCTION

1.1 BACKGROUND & RELATED WORK

Reinforcement learning for code generation has evolved through several approaches, beginning with CodeRL (Le et al., 2022) applying the REINFORCE algorithm for offline training with program-level rewards. PPOCoder (Shojaee et al., 2023) improved upon this by adopting PPO optimization with online training, while RLTF (Liu et al., 2023) eliminated the value model and combined coarse program-level rewards with fine-grained error-line feedback. StepCoder (Dou et al., 2024) combined PPO with curriculum learning over partial solutions and updated only executed tokens. SRPO (Zhang et al., 2025), built on GRPO, analyzed both mathematical and coding tasks, observing that mathematical problems tend to increase output length while coding problems tend to decrease it, though our findings indicate that newer model generations like Qwen-3 exhibit length growth tendencies even for coding tasks. The field has been supported by open-source projects including DeepCoder (Luo et al., 2025) and OlympicCoder (Penedo et al., 2025), alongside integrated training datasets such as APPS (Hendrycks et al., 2021), CodeContests (Li et al., 2022), Taco (Li et al., 2023), KodCode (Xu et al., 2025), and rStar-Coder (Liu et al., 2025a).

Scaling inference time through extended reasoning enables models to solve increasingly challenging problems, while reinforcement learning has proven effective at activating reasoning capabilities and knowledge acquired during pretraining, even without instruction fine-tuning (DeepSeek-AI, 2025). Group Relative Policy Optimization (GRPO) (Shao et al., 2024) has gained attention by eliminating the value model requirement, instead sampling multiple responses per problem and computing relative advantages. Recent GRPO algorithmic improvements have been primarily validated on mathematical reasoning tasks, addressing specific optimization limitations. Dr. GRPO (Liu et al., 2025b) identified that GRPO encourages shorter correct responses and longer incorrect ones, leading to modifications

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

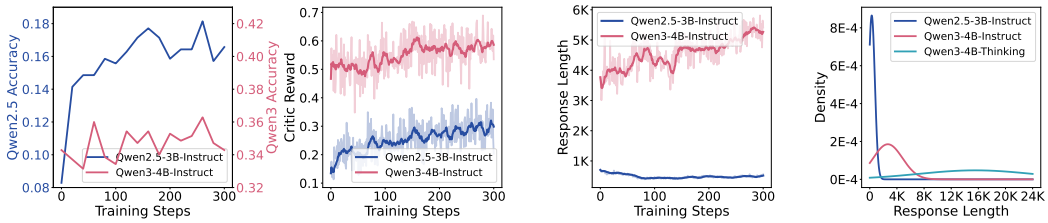


Figure 1: **Cross-Model Training Dynamics**. Performance and response length across Qwen 2.5 and Qwen 3 models, illustrating generation-specific training behaviors and output characteristics.

that remove token-level averaging in loss computation and reward standard deviation normalization in advantage calculation, thus limiting output length growth. DAPO (Yu et al., 2025) improved solution diversity by eliminating KL loss and employing high clipping ratios. Polaris (An et al., 2025) analyzed the effects of different training components such as temperature and dataset difficulty on mathematical task training. However, research on GRPO applications to coding tasks remains relatively limited compared to mathematical reasoning domains.

1.2 Motivation

Reinforcement learning insights for code generation differ from those for mathematical tasks, as coding problems require passing all test cases with additional conditions such as runtime limitations, making them more challenging and complex. Furthermore, previously accumulated training insights and datasets for traditional models often prove ineffective for modern models that have extended outputs and high reasoning abilities. Therefore, effective training of modern coding models requires updated and more challenging datasets of higher quality, new algorithms that unlock long output potential, and fresh training insights.

For example, training with GRPO on the mainstream DeepCoder dataset shows substantial improvements in Qwen 2.5 models (Qwen et al., 2025) but minimal improvements in Qwen 3 models (Yang et al., 2025), as shown in Figure 1. Critic reward analysis reveals that mainstream datasets pose greater difficulty for Qwen 2.5 while appearing relatively simple for Qwen 3 capabilities. Output behaviors reveal generational differences. Qwen 3 models exhibit pronounced upward trends in response length during training, whereas Qwen 2.5 models show stable or decreasing lengths. Additionally, across the model series progression, standard model outputs show increasing length and variance from Qwen 2.5-Instruct to Qwen 3-Instruct to Qwen 3-Thinking.

1.3 Contributions

This paper makes four contributions to improve reinforcement learning for coding models.

- **Algorithmic Innovation**: We propose MicroCoder-GRPO, extending GRPO with two original innovations, conditional truncation masking and diversity-determined temperature selection, and removal of KL loss with high clipping following DAPO. This approach achieves up to 17.6% relative improvement over strong baselines on LiveCodeBench v6, demonstrating robust performance gains across multiple model scales.
- **Systematic Analysis**: Through over thirty controlled experiments, we provide comprehensive analysis of important training components including dataset quality, code evaluators, temperature dynamics, context length and extension, truncation masking strategies, batch size and on-policy, and KL and clip ratio, offering detailed insights into their effects on reinforcement learning for code generation.
- **Dataset Creation**: We introduce MicroCoder-Dataset, a higher-quality and more challenging training corpus that achieves 3× larger performance gains than the DeepCoder dataset on LiveCodeBench v6 within 300 training steps, demonstrating effectiveness for developing coding capabilities in modern language models.
- **Infrastructure Development**: We design MicroCoder-Evaluator, a robust evaluation framework that reduces disagreement with human labels from 10.8% to 1.4% compared to the LiveCodeBench code evaluator while achieving around 40% faster execution per training step through optimized parallel processing, enabling more reliable training feedback and improved computational efficiency.

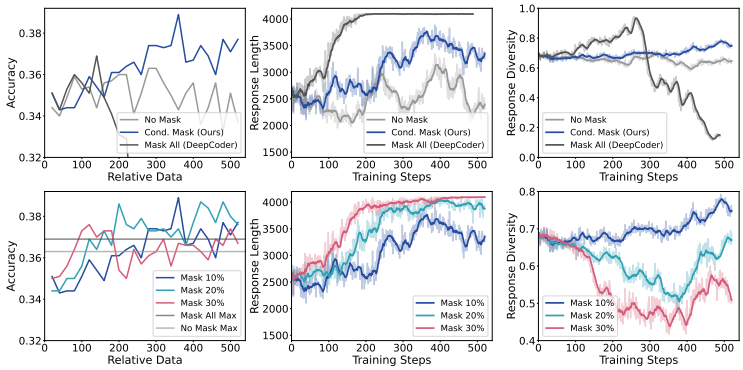


Figure 2: **Truncation Masking Effects on Training.** Performance trends under different masking strategies, comparing no mask, complete masking, and conditional masking at various rates.

2 Algorithms

2.1 Preliminaries: GRPO

Group Relative Policy Optimization (GRPO) (Shao et al., 2024) reduces reinforcement learning training costs by eliminating the separate value model typically required in policy gradient methods. For each query q, GRPO samples a group of G outputs $\{o_1, o_2, \ldots, o_G\}$ from the old policy $\pi_{\theta_{\text{old}}}$ and optimizes the current policy π_θ by maximizing $\mathcal{J}_{\text{GRPO}}(\theta)$. The advantage function A_i employs group-based normalization to estimate relative output quality. GRPO is shown as the black components in Equations 1 and 2.

2.2 MicroCoder-GRPO

We propose MicroCoder-GRPO, which introduces two original modifications to standard GRPO and one following DAPO. First, conditional truncation masking selectively zeros advantage scores for responses that simultaneously reach maximum length $L_{\max}$, produce non-incorrect answers, avoid repetition sequences (final 128 tokens differ from preceding 128 tokens), and are randomly selected with probability ρ. Second, diversity-determined temperature control updates the sampling temperature $T(D)$ from the measured response diversity toward a calibrated target $D^\star$ by Equations 3 and 4, preventing the sustained diversity decline that causes training failure. Third, following DAPO, we remove KL loss ($\beta_0 = 0$) and employ high clipping ($\varepsilon_{\text{high}}$) to maintain output diversity and response length growth. The modifications of MicroCoder-GRPO compared to GRPO are shown as the red components in Equations 1–4, where θ and θ_{old} represent current and old policy parameters, π_{ref} is the reference model, ε controls the clipping trust region, β_0 weights KL loss, r_i denotes the reward for output o_i, non-incorrect(o_i) holds if o_i contains no complete code block or its extracted program passes all test cases, repeat(o_i, m) is true when the final m tokens of o_i equal the preceding m tokens, ρ controls masking probability, $U(0,1)$ is uniform distribution over $[0,1]$, and $\mathbb{I}[\cdot]$ is the indicator function. The symbols of Equations 3 and 4 are defined in Section 2.4.

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}_{q\sim P(Q)}\left[\sum_{i=1}^{G}\left(\min\left(\frac{\pi_\theta^{T(D)}(o_i|q)}{\pi_{\theta_{\text{old}}}^{T(D)}(o_i|q)}A_i, \text{clip}\left(\frac{\pi_\theta^{T(D)}(o_i|q)}{\pi_{\theta_{\text{old}}}^{T(D)}(o_i|q)}, 1-\varepsilon, 1+\varepsilon_{high}\right)A_i\right) - \beta_0 \mathbf{D}_{\text{KL}}\left(\pi_\theta \| \pi_{\text{ref}}\right)\right)\right] \tag{1}$$

$$A_i = \frac{r_i - \text{mean}(\{r_1, r_2, \ldots, r_G\})}{\text{std}(\{r_1, r_2, \ldots, r_G\})} \cdot \left(1 - \mathbb{I}\left[|o_i| = L_{\max} \wedge \text{non-incorrect}(o_i) \wedge \neg\text{repeat}(o_i, m) \wedge U(0,1) < \rho\right]\right) \tag{2}$$

$$D_t = \frac{1}{|B_t|}\sum_{q\in B_t}\frac{\left|\text{uniq}_4\left(o_1^q, \ldots, o_G^q\right)\right|}{\sum_{i=1}^{G}\left|\text{gram}_4\left(o_i^q\right)\right|}, \qquad \bar{D}_t = (1-\lambda)\,\bar{D}_{t-1} + \lambda\, D_t, \qquad D^\star = \underset{T:\,\text{stable}(T)}{\text{mean}}\ \underset{t>S-N}{\text{mean}}\ \bar{D}_t^{(T)} \tag{3}$$

$$T_t = T(\bar{D}_{t-1}) = T_{t-1}\cdot\text{clip}\left(\left(\frac{1-\bar{D}_{t-1}}{1-D^\star}\right)^{\eta/\alpha}, 1-\delta, 1+\delta\right), \qquad T_0 = \max\{\,T : D_0^{(T)} \le D^\star\,\} \tag{4}$$

2.3 Conditional Truncation Mask

Truncation masking improves long output generation potential by setting advantage scores to zero for responses that reach maximum length limits, preventing truncated outputs from contributing to policy optimization. Conditional truncation masking applies selective criteria, only masking responses that simultaneously reach maximum length, achieve non-incorrect answers, avoid repetition sequences (final 128 tokens differ from preceding 128 tokens), and masks only a specified proportion rather than all qualifying responses. As shown in Figure 2, higher masking rates accelerate output length growth and make convergence values closer to maximum response limits, with 30% masking achieving

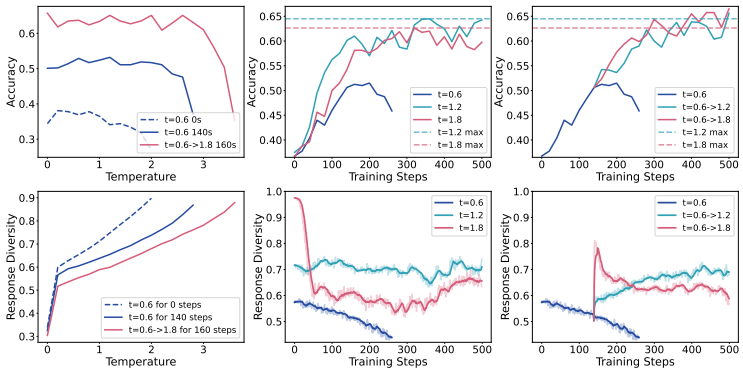


Figure 3: **Temperature Dynamics and Scheduling Analysis.** Temperature robustness trends during training showing increasing stability at higher temperatures, convergence of output diversity across different temperature settings, training failure when low temperatures cause initial diversity to fall below convergence values, and better performance of dynamic temperature scheduling (low-to-high transition) compared to static temperature approaches.

growth rates comparable to complete masking. Increased masking also accelerates response diversity decline and reduces diversity convergence values.

Masking creates distinct performance dynamics where training rapidly rises to higher values, then declines, and converges to specific performance levels. Masking proportion creates trade-offs between training speed and peak performance. Increased masking enables faster achievement of initial performance peaks, while reduced masking extends the initial improvement stage and achieves higher peak performance values. Conditional truncation masking demonstrates better training stability compared to both no masking and complete masking approaches, achieving higher final performance while avoiding the rapid training decrease observed with complete masking strategies.

2.4 Diversity-determined Temperature

Temperature scheduling influences GRPO training stability and convergence (Figure 3). Analysis reveals that models develop increasing temperature robustness throughout training, with the upper bound of stable temperatures progressively increasing. Higher temperatures naturally increase output diversity, but this diversity systematically decreases at fixed temperatures as training progresses. **Despite varying temperatures, output diversity converges to similar final values across different temperature settings.** Diversity follows Equation 3 as unique over total token 4-grams across the G responses per query, batch-averaged and smoothed by λ with $\bar{D}_0 = D_0$, where higher values indicate greater output diversity. Important temperature thresholds emerge during training. When initial output diversity falls clearly below expected convergence values, models experience continued diversity reduction accompanied by training failure. Traditionally standard temperatures ($T = 0.6$) can cause training failure, while modern models like Qwen-3 demonstrate stable training even at elevated temperatures ($T = 1.8$) with minimal influence on final convergence values. The convergence value of response diversity thus depends on the training configuration; we take it as the target $D^\star$ of Equation 3, since sustained diversity below it fails while excessive temperature causes drastic fluctuations.

Dynamic temperature scheduling yields better performance compared to static temperature approaches. The optimal method employs initial low-temperature training followed by high-temperature stages. This approach reduces initial output diversity during the high-temperature stage, ultimately achieving better performance than direct high-temperature training from initialization. However, we observe that continuous uniform temperature changes influence training stability, and even brief continuous temperature increases or decreases within a small number of steps can cause irreversible change in output diversity. MicroCoder-GRPO therefore controls diversity in closed loop by Equation 4, raising the temperature every τ steps if the smoothed diversity is below $D^\star$ and lowering it otherwise. The step follows the empirical power law $1 - D \propto T^{-\alpha}$, whose exponent is fixed during training while its prefactor grows as the policy sharpens, so one measurement fixes the update; η damps and δ bounds it. $D^\star$ is calibrated once per configuration from constant-temperature runs on a grid from $T_{\min}$ to $T_{\max}$, while T_0 and α use $D_0^{(T)}$ measured by sampling the first training batch at each grid temperature. stable(T) holds if the mean training reward over the last N of S steps is not

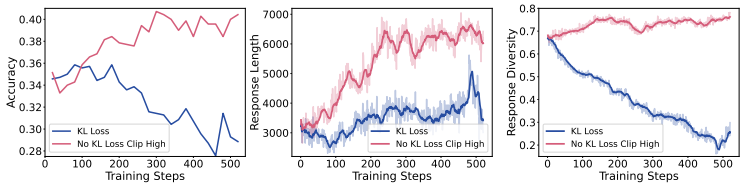


Figure 4: **Influence of KL Loss and Clip Ratio on Training Dynamics**. Performance comparison between standard KL loss and removed KL loss with high clipping, illustrating the relationship between diversity maintenance and sustained performance improvement.

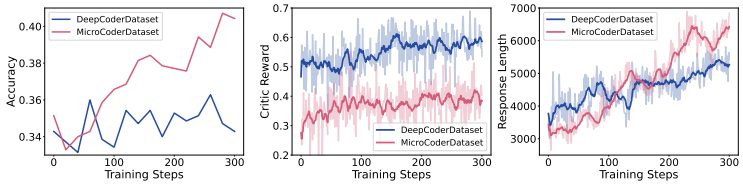


Figure 5: **Dataset Quality Comparison.** Training dynamics comparing MicroCoder and DeepCoder datasets across accuracy, critic reward, and response length metrics, demonstrating learning effectiveness on challenging problems.

below that over the first N; $D^\star$ averages the last N steps of the stable runs by Equation 3; T_0 is the highest grid temperature with $D_0^{(T)} \leq D^\star$, so the temperature warms up from below the target; α is the least-squares log-log slope of $1 - D_0^{(T)}$ against T over the three grid points centered on T_0. Section 5 gives all values.

2.5 No KL Loss and High Clip Ratio

Removing KL loss with high clipping improves output diversity and response length, driving sustained performance improvements (Figure 4). Standard KL loss without high clipping reduces output diversity and limits response length to marginal increases, causing initial performance gains followed by decline. Continued diversity reduction creates unsustainable training dynamics where performance first rises then falls, preventing effective long-term training.

3 Data

Our data processing framework employs a four-stage pipeline to create high-quality coding datasets, detailed in Appendix F with the prompts in Appendix E. The Collect stage collects data from diverse sources to maximize coverage. The Process stage standardizes data through language translation, noise removal, format normalization, and completeness validation. The Filter stage applies multi-criteria selection based on textual quality, content relevance, and difficulty distribution. Finally, the Verify stage conducts validation to ensure problem readability, completeness, and test case accuracy. This end-to-end pipeline transforms raw datasets into a high-quality, standardized corpus suitable for reinforcement learning training. **All problems comply with the copyright terms of their sources, and the entire dataset, including the privately collected portion, has been released on open-source platforms.** A complete released record is shown in Appendix G.

Comparative analysis reveals that MicroCoder dataset drives better coding ability improvements over DeepCoder dataset (Figure 5). Training on MicroCoder dataset yields rapid, obvious accuracy gains, while DeepCoder dataset training shows minimal performance variation. This performance differential correlates with inherent dataset difficulty. MicroCoder dataset consistently generates lower critic rewards, indicating higher problem complexity. Despite both datasets exhibiting similar critic reward growth trends during training, only MicroCoder dataset produces test set improvements, demonstrating that training effectiveness on challenging problems translates more directly to generalization performance. Table 1 presents comprehensive benchmark results comparing mainstream and MicroCoder datasets under identical training configurations.

Harder problems exhibit accelerated response length growth with greater final magnitudes. Although MicroCoder dataset initially produces similar or shorter response lengths compared to DeepCoder dataset, it demonstrates faster growth rates and ultimately achieves longer outputs. This indicates that challenging coding tasks require longer solution paths.

Table 1: **Performance Evaluation Across Model Scales and Dataset Sources.** MicroCoder consistently outperforms baselines across model scales, benchmarks, and difficulty levels, with the largest gains on medium and hard problems. Delta rows are measured against the baseline with the highest overall LiveCodeBench accuracy in each block.

	AtCoder				LeetCode				LiveCodeBench			
	Easy	Medium	Hard	All	Easy	Medium	Hard	All	Easy	Medium	Hard	All
Dataset Comparison												
GRPO, Qwen3 4B, 8K												
APPS	96.2	36.5	11.3	36.8	79.4	14.4	1.3	27.8	89.5	25.5	8.7	33.6
DeepCoder	96.2	40.4	10.4	37.3	73.5	19.2	0.0	27.8	87.2	29.8	7.8	33.9
MicroCoder (Ours)	97.1	47.1	11.3	**39.5**	76.5	18.3	3.7	**29.4**	89.0	32.7	9.4	**35.9**
Delta	+0.9	+6.7	+0.9	+2.2	+3.0	-0.9	+3.7	+1.6	+1.8	+2.9	+1.6	+2.0
DAPO, Qwen3 4B, 8K												
APPS	99.0	39.4	12.9	39.1	79.4	24.0	2.5	32.1	91.3	31.7	10.3	36.6
CodeContests	98.1	44.2	11.3	39.1	76.5	20.2	0	29.0	89.5	32.2	8.4	35.4
KodCode RL	99.0	46.2	11.3	39.7	77.9	26.0	2.5	32.5	90.7	36.1	9.1	37.1
KodCode Instruct	98.1	46.2	10.0	38.8	83.8	23.1	5.0	33.7	92.4	34.6	8.7	37.0
DeepCoder	98.1	40.4	12.1	38.6	77.9	24.0	3.7	32.1	90.1	32.2	10.0	36.3
MicroCoder (Ours)	99.0	49.0	14.6	**42.2**	83.8	33.7	5.0	**38.1**	93.0	41.3	12.2	**40.7**
Delta	+0	+2.8	+3.3	+2.5	+5.9	+7.7	+2.5	+5.6	+2.3	+5.2	+3.1	+3.6
Model Ablation												
DAPO, DeepSeek-R1-Distill-Llama-8B, 16K												
APPS	98.1	25.0	2.9	30.1	92.6	22.1	3.7	35.3	95.9	23.6	3.1	32.0
CodeContests	100	34.6	2.5	32.6	88.2	26.0	2.5	35.3	95.3	30.3	2.5	33.6
KodCode RL	100	21.2	2.5	29.5	91.2	26.0	1.3	35.7	96.5	23.6	2.2	31.7
DeepCoder	100	33.7	7.1	34.8	85.3	25.0	1.3	33.7	94.2	29.3	5.6	34.4
MicroCoder (Ours)	98.1	39.4	7.5	**35.9**	92.6	26.9	5.0	**37.7**	95.9	33.2	6.9	**36.6**
Delta	-1.9	+5.7	+0.4	+1.1	+7.3	+1.9	+3.7	+4.0	+1.7	+3.9	+1.3	+2.2
DAPO, Qwen3 14B, 8K												
APPS	100	50	13.8	42.2	70.6	20.2	5.0	29.0	88.4	35.1	11.6	37.4
CodeContests	98.1	50.0	11.3	40.4	85.3	18.3	2.5	31.3	93.0	34.1	9.1	37.1
KodCode RL	97.1	37.5	6.7	34.8	76.5	24.0	2.5	31.3	89.0	30.8	5.6	33.6
DeepCoder	99.0	51.9	13.3	42.2	83.8	26.9	5.0	35.3	93.0	39.4	11.3	39.7
MicroCoder (Ours)	99.0	50.0	17.5	**44.0**	91.2	31.7	5.0	**39.3**	95.9	40.9	14.4	**42.3**
Delta	+0	-1.9	+4.2	+1.8	+7.4	+4.8	+0	+4.0	+2.9	+1.5	+3.1	+2.6
Component Ablation												
GRPO, Qwen3 1.7B, 4K												
OlympicCoder Subset	73.1	16.3	3.7	22.8	48.5	3.8	0	14.7	63.4	10.1	2.8	19.9
DeepCoder Subset	75.0	20.2	1.7	23.0	41.2	3.8	0	12.7	61.6	12.0	1.3	19.3
MicroCoder Subset	83.7	16.3	0.4	**23.4**	54.4	3.8	0	**16.3**	72.1	10.1	0.3	**20.9**
DAPO, Qwen3 1.7B, 4K												
OlympicCoder Subset	83.7	13.5	0	22.5	47.1	3.8	0	14.3	69.2	8.7	0	19.6
DeepCoder Subset	78.8	15.4	0.8	22.3	50.0	3.8	0	15.1	67.4	9.6	0.6	19.7
MicroCoder Subset	78.8	21.2	0	**23.2**	61.8	7.7	0	**19.8**	72.1	14.4	0	**22.0**
Filtering Analysis												
DAPO, Qwen3 8B, 8K												
OlympicCoder	93.3	38.5	6.2	33.9	72.1	17.3	1.3	27.0	84.9	27.9	5.0	31.4
OlympicCoder Filtered	92.3	42.3	9.6	**36.4**	64.7	19.2	6.2	**27.4**	81.4	30.8	8.7	**33.1**
KodCode	88.5	40.4	5.0	32.6	69.1	13.5	1.3	24.6	80.8	26.9	4.1	29.7
KodCode Filtered	89.4	40.4	7.9	**34.4**	69.1	18.3	0	**26.2**	81.4	29.3	5.9	**31.4**

4 INFRASTRUCTURE

MicroCoder-Evaluator implements comprehensive output validation through multi-method comparison with 6-7 fallback methods, format flexibility handling lists, tuples, strings, sets with automatic type conversions, approximate numeric comparison using np.allclose() for floating point tolerance, preprocessing including multi-line splitting and whitespace normalization, and high fault tolerance that continues attempting different comparison approaches when individual methods fail. In contrast, LiveCodeBench Evaluator employs exact matching through direct equality comparison, precise numerics via Decimal library for floating point calculations, and minimal preprocessing limited to basic whitespace stripping.

The comprehensive evaluation approach yields better training outcomes (Figure 6). On 500 model outputs on MicroCoder-Dataset problems labeled by three annotators, MicroCoder-Evaluator disagrees with the human labels on 1.4% against 10.8% for the LiveCodeBench evaluator, which rejects roughly one in five correct outputs over format variants, and neither evaluator accepts any of 1,000 near-miss outputs that match the expected format but are semantically wrong; Appendix H gives the breakdown and Appendix I worked examples. MicroCoder-Evaluator achieves higher critic reward scores, indicating more accurate assessment of solution quality. This translates to improved model training effectiveness with fewer misjudgments, reduced noise, and accelerated and higher test accuracy improvement. The performance differential between evaluators is particularly pronounced during early training stages, where robust evaluation becomes important for establishing proper learning feedback and preventing suboptimal convergence.

5 EXPERIMENTAL DESIGN

For algorithmic comparisons, we evaluate MicroCoder-GRPO against GRPO and DAPO (Yu et al., 2025) that removes KL loss and employ high clipping. For temperature dynamics analysis, we employ the OlympicCoder training set with evaluation on 200 randomly selected problems from the test set using maximum response length of 4K tokens. All other analyses and model evaluations utilize DeepCoder (Luo et al., 2025) and MicroCoder datasets with testing conducted on all the unseen questions in the whole LiveCodeBench v6, which includes AtCoder and LeetCode but no Codeforces problems in this version. Experiments are performed using Qwen3-1.7B Instruct, Qwen3-4B-Instruct-2507, Qwen3-8B, Qwen3-14B (Yang et al., 2025), and DeepSeek-R1-Distill-Llama-8B (DeepSeek-AI, 2025) models to demonstrate method robustness across different model scales. Unless otherwise specified, default experimental configurations employ MicroCoder dataset, LiveCodeBench



Figure 6: **Training Performance with Different Code Evaluators.** Accuracy and reward during training using MicroCoder-Evaluator versus LiveCodeBench Evaluator, demonstrating the benefits of robust output validation for coding task training. The right subfigure shows the efficiency improvements of MicroCoder-Evaluator through parallel processing optimization compared to the original DeepCoder single-threaded version.

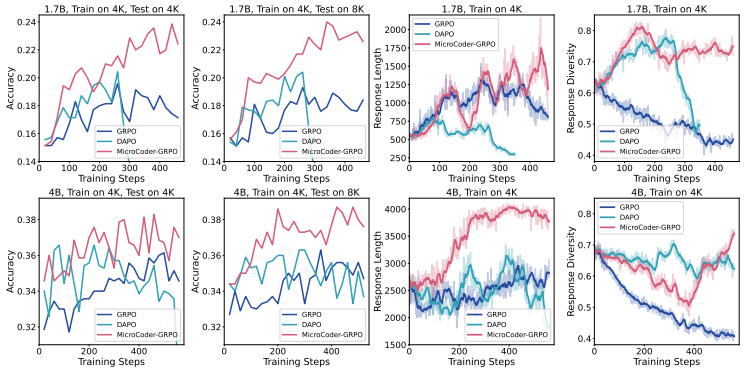


Figure 7: **Performance comparison** between MicroCoder-GRPO and baseline approaches across different model scales and output lengths, illustrating training accuracy, response length, and output diversity dynamics, demonstrating the better stability and sustained improvements of the proposed method under both preset and extended context lengths.

v6 evaluation, Qwen3-4B-Instruct-2507 model, maximum response length of 8K tokens, temperature of 1.2 for GRPO and DAPO, train batch size of 64, learning rate of 1e-6, DAPO algorithm, 8 samples per query during training with 0-1 binary accuracy as reward, and evaluation based on average accuracy across three training seeds, 0, 42, and 50, with four inference attempts each; standard deviations are given in Appendix A. MicroCoder-GRPO uses $\rho = 0.1$, $m = 128$, Qwen3-tokenizer 4-grams for all models, and Equation 4 with $\lambda = 0.1$, $\eta = 0.5$, $\delta = 0.2$, $\tau = 30$; calibration sweeps $T = 0.6$ to 1.8 in steps of 0.6 for $S = 300$ steps with $N = 50$ to obtain $D^\star$, while α and T_0 are measured by sampling the initial policy on a grid from 0.6 to 1.8 in steps of 0.2, giving $D^\star = 0.72$, $\alpha = 0.8$, $T_0 = 1.2$. Across the five models, $D^\star$ averages 0.73 with a standard deviation of 0.02; a difference of 0.02 in $D^\star$ shifts the target temperature by about 9% under Equation 4, well within one grid step, so a single value is used throughout. Hence $D^\star$ calibrated on one model can be reused across model sizes and families without recalibration. Calibration costs three runs of 300 steps plus inference-only sampling, comparable to an ordinary hyperparameter search, and is performed once per configuration. The clipped policy update uses $\varepsilon = 0.2$, $\varepsilon_{\text{high}} = 0.28$ for DAPO and MicroCoder-GRPO, mini-batch 32, AdamW, gradient clipping 1.0, no entropy penalty, KL weight $\beta_0 = 0.001$ for GRPO only, maximum prompt length 2048, and 520 training steps, on verl with vLLM sampling over the full vocabulary, top-p 1.0, top-k off, and otherwise default settings, on four nodes with eight A100 GPUs each. **MicroCoder-Evaluator *only* computes *training* rewards; all reported test accuracies use the *official* LiveCodeBench v6 code, evaluator, and default generation parameters at the stated maximum response length, so training rewards cannot inflate them. Manual inspection of 1,000 randomly sampled positively rewarded training trajectories found no reward hacking.**

6 Results

Table 2 varies *only* the algorithm and Table 1 *only* the dataset, and Figures 2, 3, 4, and 6 each vary *one* component. Experimental evaluation compares training dynamics under 4K response length with testing on both 4K and 8K contexts to assess algorithmic effectiveness and reasoning budget scalability (Figure 7). MicroCoder-GRPO demonstrates better performance across both testing configurations, achieving faster convergence rates and higher final accuracy values compared

Table 2: **Quantitative evaluation results** across different datasets comparing MicroCoder-GRPO against baseline methods on various model sizes, difficulty levels, and output budgets to assess algorithmic improvements and extended reasoning capabilities. ***Only* the algorithm differs within each block; dataset, evaluator, and hyperparameters are fixed.** Delta rows are measured against the baseline with the highest overall LiveCodeBench accuracy in each block.

	AtCoder				LeetCode				LiveCodeBench			
	Easy	Medium	Hard	All	Easy	Medium	Hard	All	Easy	Medium	Hard	All
Not trained												
Qwen3-1.7B Instruct (4K)	69.2	10.6	1.3	19.2	33.8	3.8	0	10.7	55.2	7.2	0.9	16.1
Qwen3-1.7B Instruct (8K)	69.2	10.6	1.3	19.2	33.8	3.8	0	10.7	55.2	7.2	0.9	16.1
Qwen3-4B-Instruct (4K)	95.2	44.2	9.2	37.3	73.5	19.2	2.5	28.6	86.6	31.7	7.5	34.1
Qwen3-4B-Instruct (8K)	95.2	45.2	10.0	37.9	76.5	18.3	2.5	29.0	87.8	31.7	8.1	34.7
Qwen3 1.7B, Train on 4K, Test on 4K												
GRPO	70.2	20.2	1.7	21.9	47.1	**4.8**	**0**	14.7	61.0	12.5	1.3	19.3
DAPO	76.9	15.4	2.9	23.0	55.9	1.9	**0**	15.9	68.6	8.7	**2.2**	20.4
MicroCoder-GRPO	**83.7**	**24.0**	**2.9**	**26.6**	**57.4**	**4.8**	**0**	**17.5**	**73.3**	**14.4**	**2.2**	**23.3**
Δ	+6.8	+8.6	0.0	+3.6	+1.5	+2.9	0	+1.6	+4.7	+5.7	0.0	+2.9
Qwen3 1.7B, Train on 4K, Test on 8K												
GRPO	70.2	20.2	1.7	21.9	47.1	4.8	**0**	14.7	61.0	12.5	1.3	19.3
DAPO	76.9	15.4	2.9	23.0	**55.9**	1.9	**0**	15.9	68.6	8.7	2.2	20.4
MicroCoder-GRPO	**82.7**	**26.9**	**3.7**	**27.5**	52.9	**8.7**	**0**	**17.9**	**70.9**	**17.8**	**2.8**	**24.0**
Δ	+5.8	+11.5	+0.8	+4.5	-3.0	+6.8	0	+2.0	+2.3	+9.1	+0.6	+3.6
Qwen3 4B, Train on 4K, Test on 4K												
GRPO	98.1	46.2	**11.7**	**39.7**	72.1	21.2	0	28.2	87.8	33.7	**8.7**	35.6
DAPO	97.1	**49.0**	10.0	39.3	75.0	**22.1**	1.3	29.8	88.4	**35.6**	7.8	35.9
MicroCoder-GRPO	**100.0**	**49.0**	9.6	**39.7**	**88.2**	**22.1**	**3.7**	**34.1**	**95.3**	**35.6**	8.1	**37.7**
Δ	+2.9	0.0	-0.4	+0.4	+13.2	0.0	+2.4	+4.3	+6.9	0.0	+0.3	+1.8
Qwen3 4B, Train on 4K, Test on 8K												
GRPO	**100.0**	45.2	**12.9**	40.6	72.1	22.1	0	28.6	89.0	33.7	9.7	36.3
DAPO	**100.0**	43.3	11.3	39.3	76.5	23.1	2.5	31.0	90.7	33.2	9.1	36.3
MicroCoder-GRPO	**100.0**	**48.1**	**12.9**	**41.3**	**83.8**	**24.0**	**5.0**	**34.1**	**93.6**	**36.1**	**10.9**	**38.7**
Δ	0.0	+4.8	+1.6	+2.0	+7.3	+0.9	+2.5	+3.1	+2.9	+2.9	+1.8	+2.4
DeepSeek-R1-Distill-Llama-8B, Train on 16K, Test on 16K												
GRPO	**100.0**	39.4	5.8	35.5	**89.7**	26.0	1.3	35.3	**95.9**	32.7	4.7	35.4
DAPO	**100.0**	**44.2**	8.3	37.9	88.2	25.0	2.5	34.9	95.3	34.6	6.9	36.9
MicroCoder-GRPO	**100.0**	43.3	**10.0**	**38.6**	**89.7**	**32.7**	**8.7**	**40.5**	**95.9**	**38.0**	**9.7**	**39.3**
Δ	0	-0.9	+1.7	+0.7	+1.5	+7.7	+6.2	+5.6	+0.6	+3.4	+2.8	+2.4

to baselines. Additionally, our 4K context training achieves performance comparable to baseline methods trained with 6K contexts, while saving approximately 40-50% computational cost due to the $O(n^2)$ of self-attention. While DAPO with removed KL loss and high clipping reaches higher peak performance than standard GRPO and achieves peak values more rapidly, it exhibits training variability with pronounced performance decrease during extended training stages. In contrast, MicroCoder-GRPO's conditional truncation masking not only accelerates improvement and improves convergence values but maintains stable long-term training dynamics without the failure observed in DAPO. Analysis reveals that 4B models exhibit greater response length growth capacity compared to 1.7B models, with MicroCoder-GRPO producing length increases across both model scales while preserving output diversity and sustained accuracy improvements throughout training.

MicroCoder-GRPO outperforms both GRPO and DAPO on nearly all benchmark datasets, difficulty levels, and model sizes, demonstrating robust improvements in coding task performance (Table 2). The same holds when training on DeepCoder, as Table 5 in Appendix C shows. The performance gains become more pronounced under extended context evaluation, where models trained on 4K contexts are tested on 8K contexts, with 1.7B models achieving +3.6% improvement on LiveCodeBench, indicating better scalability of the proposed approach. Harder problems exhibit greater performance gains under extended output budgets, with MicroCoder-GRPO showing strong advantages on Medium and Hard difficulty levels, suggesting improved capability for complex reasoning tasks requiring longer solution development. According to the ablation study in Appendix B, conditional truncation masking, diversity-determined temperature, and KL removal with high clipping account for 33.33%, 52.38%, and 14.29% of the gain over GRPO averaged over Qwen3 models; on Qwen3-4B with 4K testing, for example, adding each alone yields 36.4, 36.7, and 35.9 in overall LiveCodeBench v6 accuracy, against 35.6 for GRPO and 37.7 for MicroCoder-GRPO. Removing any single component lowers accuracy significantly, with temperature contributing the most. Dropping the probability or non-incorrect masking condition, replacing the controller with the fixed schedule of Figure 3, or lowering $D^\star$ by 0.05 also lowers accuracy significantly.

7 Analysis

7.1 Batch Size and On-Policy

Training batch configuration influences on-policy versus off-policy learning characteristics. The framework employs train_batch_size for simultaneous problem inference and ppo_mini_batch_size for individual parameter updates, executing train_batch_size/ppo_mini_batch_size update iterations per training step cycle. Smaller train_batch_size values (maintaining constant ppo_mini_batch_size) create more on-policy behavior resembling immediate problem-solving reflection, while larger values produce off-policy dynamics akin to batch reflection after completing all problems. As shown in Figure 8, on-policy configurations exhibit reduced training stability with accelerated response

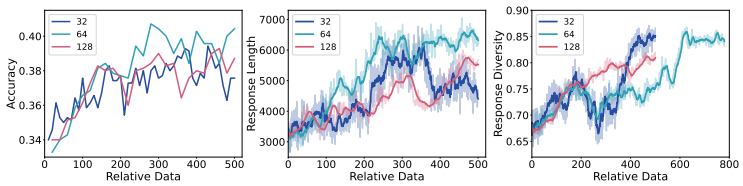


Figure 8: **On-Policy versus Off-Policy Training Effects**. Performance comparison across different train batch size settings, demonstrating that off-policy configurations increase training stability while intermediate settings achieve optimal performance.

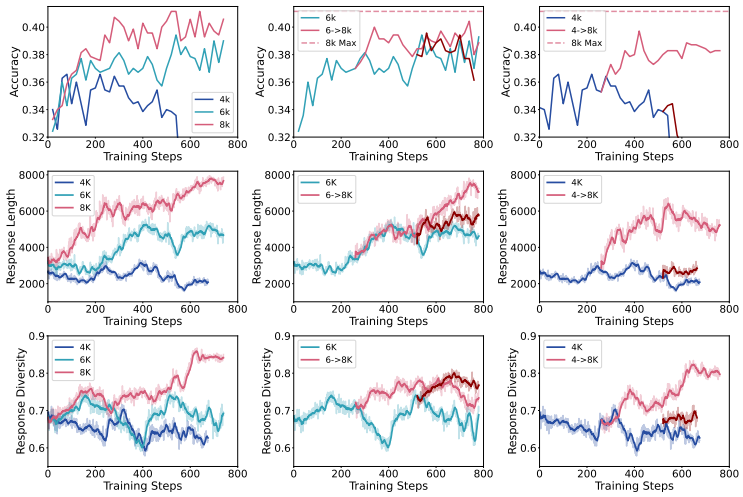


Figure 9: **Context Length Influence on Training Dynamics.** Performance trends across different maximum output length settings showing accuracy, response length growth, and output diversity, demonstrating irreversible effects of early-stage length limitations on model capabilities.

diversity convergence and response length trends that rise and then decline, whereas off-policy approaches demonstrate greater stability across both metrics. Optimal performance emerges from intermediate configurations balancing on-policy and off-policy characteristics, outperforming heavily skewed settings in either direction.

7.2 Context Length and Extension

Context length settings exhibit scaling relationships with model performance (Figure 9). Longer maximum output lengths correlate with higher final accuracy, faster output growth rates, greater final output lengths, and increased output diversity. However, small initial maximum output lengths reduce both output generation and diversity, creating persistent performance effects even after subsequent length extensions. The magnitude of these effects increases with more limiting initial settings and longer training time. Beyond specific training thresholds, models show minimal recovery when limitations are later relaxed, indicating that early-stage output reduction changes learning paths and cannot be compensated by later context extension.

8 Conclusions

This paper proposes MicroCoder-GRPO, which addresses the training bottlenecks of modern code generation models with conditional truncation masking for stable long-output growth, diversity-determined temperature for sustained output diversity, and KL removal with high clipping for solution diversity. We further introduce a more challenging dataset and a more robust and faster code evaluator, and distill over thirty controlled experiments into training insights, summarized in Appendix D, that provide systematic guidance for reinforcement learning in code generation.

Future work will focus on applying our methods and insights to diverse coding tasks, fully leveraging the performance improvements, high efficiency, and strong generalization capabilities demonstrated in this work. The systematic approach established here opens new possibilities for advancing reinforcement learning across various code generation domains.

AI USE STATEMENT

We used AI tools mainly to polish the grammar and improve the readability of the manuscript. Beyond writing, large language models were used only inside the data pipeline of Appendix F, to process coding questions and to score problem difficulty. We have reviewed all text and code.

ETHICS STATEMENT

There are no ethics concerns in this work.

REPRODUCIBILITY STATEMENT

The main text and the appendix provide the details needed to reproduce our method.

REFERENCES

Chenxin An, Zhihui Xie, Xiaonan Li, Lei Li, Jun Zhang, Shansan Gong, Ming Zhong, Jingjing Xu, Xipeng Qiu, Mingxuan Wang, and Lingpeng Kong. POLARIS: A Post-Training Recipe for Scaling Reinforcement Learning on Advanced Reasoning Models, 2025. URL https://hkunlp.github.io/blog/2025/Polaris.

Benjamin Bloom. Bloom's taxonomy, 1956.

DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025. URL https://arxiv.org/abs/2501.12948.

Shihan Dou, Yan Liu, Haoxiang Jia, Enyu Zhou, Limao Xiong, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, Zhiheng Xi, Yuhao Zhou, Tao Ji, Rui Zheng, Qi Zhang, Tao Gui, and Xuanjing Huang. StepCoder: Improving code generation with reinforcement learning from compiler feedback. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4571–4585, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.251. URL https://aclanthology.org/2024.acl-long.251/.

Maurice H Halstead. *Elements of Software Science*. Elsevier Science Inc., 1977.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring Coding Challenge Competence With APPS. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021. URL https://openreview.net/forum?id=sD93GOzH3i5.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=chfJJYC3iL.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Hoi. CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (eds.), *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=WaGvb7OzySA.

Rongao Li, Jie Fu, Bo-Wen Zhang, Tao Huang, Zhihong Sun, Chen Lyu, Guang Liu, Zhi Jin, and Ge Li. TACO: Topics in Algorithmic COde generation dataset, 2023. URL https://arxiv.org/abs/2312.14852.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven

Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, December 2022. ISSN 1095-9203. doi: 10.1126/science.abq1158. URL http://dx.doi.org/10.1126/science.abq1158.

Jiate Liu, Yiqin Zhu, Kaiwen Xiao, QIANG FU, Xiao Han, Yang Wei, and Deheng Ye. RLTF: Reinforcement Learning from Unit Test Feedback. *Transactions on Machine Learning Research*, 2023. ISSN 2835-8856. URL https://openreview.net/forum?id=hjYmsV6nXZ.

Yifei Liu, Li Lyna Zhang, Yi Zhu, Bingcheng Dong, Xudong Zhou, Ning Shang, Fan Yang, and Mao Yang. rStar-Coder: Scaling Competitive Code Reasoning with a Large-Scale Verified Dataset, 2025a. URL https://arxiv.org/abs/2505.21297.

Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. Understanding r1-zero-like training: A critical perspective. In *Second Conference on Language Modeling*, 2025b. URL https://openreview.net/forum?id=5PAF7PAY2Y.

Michael Luo, Sijun Tan, Roy Huang, Ameen Patel, Alpay Ariyak, Qingyang Wu, Xiaoxiang Shi, Rachel Xin, Colin Cai, Maurice Weber, Ce Zhang, Li Erran Li, Raluca Ada Popa, and Ion Stoica. DeepCoder: A Fully Open-Source 14B Coder at O3-mini Level, 2025. Notion Blog.

Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

Guilherme Penedo, Anton Lozhkov, Hynek Kydlíček, Loubna Ben Allal, Edward Beeching, Agustín Piqueres Lajarín, Quentin Gallouédec, Nathan Habib, Lewis Tunstall, and Leandro von Werra. Codeforces. https://huggingface.co/datasets/open-r1/codeforces, 2025.

Qwen, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, ..., and Zihan Qiu. Qwen2.5 Technical Report, 2025. URL https://arxiv.org/abs/2412.15115.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, Y.K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *CoRR*, abs/2402.03300, 2024. URL https://arxiv.org/abs/2402.03300.

Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K. Reddy. Execution-based Code Generation using Deep Reinforcement Learning. *Transactions on Machine Learning Research*, 2023. ISSN 2835-8856. URL https://openreview.net/forum?id=0XBuaxqEcG.

Zhangchen Xu, Yang Liu, Yueqin Yin, Mingyuan Zhou, and Radha Poovendran. KodCode: A diverse, challenging, and verifiable synthetic dataset for coding. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 6980–7008, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.365. URL https://aclanthology.org/2025.findings-acl.365/.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, ..., and Zihan Qiu. Qwen3 Technical Report, 2025. URL https://arxiv.org/abs/2505.09388.

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, Ru Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Yonghui Wu, and Mingxuan Wang. DAPO: An Open-Source LLM Reinforcement Learning System at Scale. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=2a36EMSSTp.

Xiaojiang Zhang, Jinghui Wang, Zifei Cheng, Wenhao Zhuang, Zheng Lin, Minglei Zhang, Shaojie Wang, Yinghan Cui, Chao Wang, Junyi Peng, Shimiao Jiang, Shiqi Kuang, Shouyu Yin, Chaohang Wen, Haotian Zhang, Bin Chen, and Bing Yu. SRPO: A Cross-Domain Implementation of

Large-Scale Reinforcement Learning on LLM, 2025. URL https://arxiv.org/abs/2504.14286.

Lexin Zhou, Lorenzo Pacchiardi, Fernando Martínez-Plumed, Katherine M. Collins, Yael Moros-Daval, Seraphina Zhang, Qinlin Zhao, Yitian Huang, Luning Sun, Jonathan E. Prunty, Zongqian Li, Pablo Sánchez-García, Kexin Jiang Chen, Pablo A. M. Casares, Jiyun Zu, John Burden, Behzad Mehrbakhsh, David Stillwell, Manuel Cebrian, Jindong Wang, Peter Henderson, Sherry Tongshuang Wu, Patrick C. Kyllonen, Lucy Cheke, Xing Xie, and José Hernández-Orallo. General Scales Unlock AI Evaluation with Explanatory and Predictive Power, 2025. URL https://arxiv.org/abs/2503.06378.

A Standard Deviations of Main Results

Table 3 reports, for the main settings of Table 2, the mean in the first row of each method and, in the second row, the standard deviation over three seeds. Under a Welch t-test with three seeds per method, the gains of MicroCoder-GRPO over DAPO in overall LiveCodeBench accuracy are significant at $p < 0.05$ for all three models.

Table 3: **Standard deviations of the main results.** Mean as in Table 2, followed by the standard deviation over three seeds.

	AtCoder				LeetCode				LiveCodeBench			
	Easy	Medium	Hard	All	Easy	Medium	Hard	All	Easy	Medium	Hard	All
Qwen3 1.7B, Train on 4K, Test on 8K												
GRPO	70.2	20.2	1.7	21.9	47.1	4.8	**0**	14.7	61.0	12.5	1.3	19.3
	±1.9	±2.5	±0.7	±0.8	±2.9	±1.7	±0.0	±1.4	±1.2	±1.3	±0.5	±0.4
DAPO	76.9	15.4	2.9	23.0	**55.9**	1.9	**0**	15.9	68.6	8.7	2.2	20.4
	±2.5	±1.7	±1.2	±1.0	±1.5	±1.0	±0.0	±0.8	±1.2	±1.0	±0.9	±0.4
MicroCoder-GRPO	**82.7**	**26.9**	**3.7**	**27.5**	52.9	**8.7**	**0**	**17.9**	**70.9**	**17.8**	**2.8**	**24.0**
	±2.5	±2.5	±1.1	±0.9	±2.9	±1.0	±0.0	±0.7	±2.7	±1.4	±0.8	±0.4
Qwen3 4B, Train on 4K, Test on 8K												
GRPO	**100.0**	45.2	**12.9**	40.6	72.1	22.1	0	28.6	89.0	33.7	9.7	36.3
	±0.0	±2.5	±0.7	±0.8	±2.9	±1.0	±0.0	±0.7	±1.2	±1.7	±0.5	±0.6
DAPO	**100.0**	43.3	11.3	39.3	76.5	23.1	2.5	31.0	90.7	33.2	9.1	36.3
	±0.0	±2.9	±0.7	±0.8	±3.9	±1.7	±1.2	±1.0	±1.5	±0.8	±0.3	±0.7
MicroCoder-GRPO	**100.0**	**48.1**	**12.9**	**41.3**	**83.8**	**24.0**	**5.0**	**34.1**	**93.6**	**36.1**	**10.9**	**38.7**
	±0.0	±1.0	±1.2	±0.6	±2.9	±1.0	±1.2	±1.4	±1.2	±1.0	±1.1	±0.7
DeepSeek-R1-Distill-Llama-8B, Train on 16K, Test on 16K												
GRPO	**100.0**	39.4	5.8	35.5	**89.7**	26.0	1.3	35.3	**95.9**	32.7	4.7	35.4
	±0.0	±1.0	±0.8	±0.4	±1.5	±2.5	±2.2	±1.4	±0.6	±1.7	±1.1	±0.6
DAPO	**100.0**	**44.2**	8.3	37.9	88.2	25.0	2.5	34.9	95.3	34.6	6.9	36.9
	±0.0	±1.0	±0.7	±0.4	±1.5	±2.9	±2.2	±1.4	±0.6	±1.7	±0.9	±0.7
MicroCoder-GRPO	**100.0**	43.3	**10.0**	**38.6**	**89.7**	**32.7**	**8.7**	**40.5**	**95.9**	**38.0**	**9.7**	**39.3**
	±0.0	±1.0	±0.8	±0.7	±2.9	±2.5	±2.2	±2.1	±1.2	±1.4	±1.1	±0.9

B Component Ablation

Table 4 ablates the three components of MicroCoder-GRPO on Qwen3-4B with 4K training and 4K testing, adding one component at a time to GRPO and removing one at a time from MicroCoder-GRPO, with mean and standard deviation over three seeds of overall accuracy. The last two blocks vary the masking conditions and the temperature control of MicroCoder-GRPO with everything else fixed; the fixed schedule is the low-to-high schedule of Figure 3, and the two $D^\star$ rows shift the target by 0.05 from its calibrated value.

Table 4: **Component ablation on Qwen3-4B.** Overall accuracy, mean ± standard deviation over three seeds; rows for GRPO, DAPO, and MicroCoder-GRPO are from Table 2.

	AtCoder	LeetCode	LiveCodeBench
Adding one component to GRPO			
GRPO	39.7 ± 0.4	28.2 ± 0.4	35.6 ± 0.3
+ KL removal with high clipping, which is DAPO	39.3 ± 0.4	29.8 ± 0.4	35.9 ± 0.4
+ conditional truncation masking	39.7 ± 0.2	30.6 ± 0.7	36.4 ± 0.3
+ diversity-determined temperature	40.0 ± 0.4	31.0 ± 0.8	36.7 ± 0.4
Removing one component from MicroCoder-GRPO			
MicroCoder-GRPO	39.7 ± 0.4	34.1 ± 0.7	37.7 ± 0.2
without KL removal with high clipping	40.0 ± 0.4	31.3 ± 0.7	36.9 ± 0.4
without conditional truncation masking	39.5 ± 0.2	32.1 ± 0.7	36.9 ± 0.4
without diversity-determined temperature	39.3 ± 0.4	31.3 ± 0.8	36.4 ± 0.5
Masking condition of MicroCoder-GRPO			
without the probability condition	39.3 ± 0.2	31.3 ± 0.4	36.4 ± 0.2
without the non-incorrect condition	39.3 ± 0.2	32.1 ± 0.7	36.7 ± 0.3
without the repetition condition	39.5 ± 0.4	33.3 ± 0.7	37.3 ± 0.5
Temperature control of MicroCoder-GRPO			
fixed low-to-high schedule 0.6 to 1.8 as in Figure 3	39.3 ± 0.2	32.5 ± 0.8	36.9 ± 0.4
$D^\star = 0.67$ instead of 0.72	39.3 ± 0.2	32.9 ± 0.8	37.0 ± 0.2
$D^\star = 0.77$ instead of 0.72	39.7 ± 0.4	33.3 ± 0.4	37.4 ± 0.2

C ALGORITHM COMPARISON ON DEEPCODER

Table 5 compares the three algorithms trained on the DeepCoder dataset with Qwen3-4B and Qwen3-14B at 8K and DeepSeek-R1-Distill-Llama-8B at 16K, holding everything except the algorithm fixed; the DAPO rows and the Qwen3-4B GRPO row are from Table 1. MicroCoder-GRPO outperforms both baselines in all three settings, so the algorithmic gain does not depend on MicroCoder-Dataset.

Table 5: **Algorithm comparison on DeepCoder.** Three models trained and tested on the DeepCoder dataset at the stated lengths; the Delta row is measured against the baseline with the highest overall LiveCodeBench accuracy.

	AtCoder				LeetCode				LiveCodeBench			
	Easy	Medium	Hard	All	Easy	Medium	Hard	All	Easy	Medium	Hard	All
Qwen3 4B on DeepCoder, Train on 8K, Test on 8K												
GRPO	96.2	40.4	10.4	37.3	73.5	19.2	0.0	27.8	87.2	29.8	7.8	33.9
DAPO	98.1	40.4	12.1	38.6	77.9	24.0	3.7	32.1	90.1	32.2	10.0	36.3
MicroCoder-GRPO	**99.0**	**42.3**	**14.6**	**40.6**	**83.8**	**26.0**	**7.5**	**35.7**	**93.0**	**34.1**	**12.8**	**38.9**
Δ	+0.9	+1.9	+2.5	+2.0	+5.9	+2.0	+3.8	+3.6	+2.9	+1.9	+2.8	+2.6
DeepSeek-R1-Distill-Llama-8B on DeepCoder, Train on 16K, Test on 16K												
GRPO	100.0	30.8	5.8	33.5	83.8	23.1	1.3	32.5	93.6	26.9	4.7	33.1
DAPO	100.0	33.7	7.1	34.8	85.3	25.0	1.3	33.7	94.2	29.3	5.6	34.4
MicroCoder-GRPO	**100.0**	**35.6**	**6.7**	**35.0**	**86.8**	**31.7**	**7.5**	**38.9**	**94.8**	**33.7**	**6.9**	**36.4**
Δ	0.0	+1.9	-0.4	+0.2	+1.5	+6.7	+6.2	+5.2	+0.6	+4.4	+1.3	+2.0
Qwen3 14B on DeepCoder, Train on 8K, Test on 8K												
GRPO	99.0	50.0	12.5	41.3	82.4	26.0	3.8	34.1	92.4	38.0	10.3	38.7
DAPO	99.0	51.9	13.3	42.2	83.8	26.9	5.0	35.3	93.0	39.4	11.3	39.7
MicroCoder-GRPO	**100.0**	**54.8**	**15.8**	**44.4**	**88.2**	**33.7**	**10.0**	**40.9**	**95.3**	**44.2**	**14.4**	**43.1**
Δ	+1.0	+2.9	+2.5	+2.2	+4.4	+6.8	+5.0	+5.6	+2.3	+4.8	+3.1	+3.4

D TRAINING INSIGHTS

This appendix lists the training insights summarized from our experiments, grouped by aspect and preceded by the observations that motivate this work.

D.1 MOTIVATION

- **Cross-generational training effectiveness.** Current training methods demonstrate substantial improvements on Qwen 2.5 models but minimal improvements on Qwen 3 models, revealing generation-specific training bottlenecks.
- **Dataset difficulty gap.** Mainstream datasets pose greater difficulty for Qwen 2.5 while appearing relatively simple for Qwen 3 capabilities, indicating need for more challenging training corpora.
- **Fundamental behavioral differences.** Output behavior patterns differ fundamentally between generations; Qwen 3 models exhibit pronounced upward trends in response length during training whereas Qwen 2.5 models show stable or decreasing lengths; across model series progression from Qwen 2.5 Instruct to Qwen 3 Instruct to Qwen 3 Thinking, standard outputs demonstrate increasing length and variance.

D.2 CODE EVALUATOR ROBUSTNESS

- **MicroCoder-Evaluator capabilities.** Multi-strategy comparison with 6-7 fallback methods, format flexibility handling lists/tuples/strings/sets with automatic type conversions, approximate numerics using np.allclose() for floating point tolerance plus rounding, extensive preprocessing including multi-line splitting, set comparison, and whitespace normalization, high fault tolerance continuing different comparison approaches when individual methods fail.
- **LiveCodeBench Evaluator capabilities.** Exact matching through direct equality comparison (prediction == gt_out), precise numerics via Decimal library for high-precision floating point comparison, minimal preprocessing limited to basic whitespace stripping.
- **Comprehensive validation benefits.** MicroCoder-Evaluator achieves higher critic reward scores, indicating more accurate assessment of solution quality.
- **Training effectiveness.** MicroCoder-Evaluator enables more effective model capability development with fewer misjudgments, reduced noise injection, faster test accuracy improvement, and higher convergence values.

- **Temporal dynamics.** Performance differential between evaluators is particularly pronounced during early training stages, where robust evaluation becomes critical for establishing proper learning feedback.

D.3 Temperature Dynamics

- **Temperature robustness.** Models develop increasing temperature robustness throughout training, with the upper bound of stable temperatures progressively expanding.
- **Temperature-diversity relationship.** Higher temperatures naturally increase output diversity.
- **Diversity decrease.** Output diversity systematically decreases at fixed temperatures as training progresses.
- **Critical diversity threshold.** When initial output diversity falls clearly below expected convergence values, models experience continued diversity reduction accompanied by training failure.
- **Traditional temperature limitations.** Conventionally standard temperatures ($T = 0.6$) can cause training failure in modern models.
- **Modern model capability.** Contemporary models like Qwen-3 demonstrate stable training even at elevated temperatures ($T = 1.8$) with minimal influence on final convergence values.
- **Convergence consistency.** Output diversity converges to similar final values across different temperature settings despite varying temperatures.
- **Dynamic scheduling advantage.** Low-to-high temperature scheduling yields superior performance by reducing initial diversity during high-temperature stages, ultimately achieving better results than direct high-temperature training from initialization.

D.4 Data Quality

- **Superior improvement effectiveness.** MicroCoder dataset drives rapid and pronounced accuracy gains, while DeepCoder dataset training shows minimal performance variation, demonstrating MicroCoder dataset's effectiveness for improving model coding capabilities.
- **Dataset difficulty.** MicroCoder dataset consistently generates lower critic rewards, indicating higher problem complexity.
- **Challenging problem effectiveness.** Despite both datasets exhibiting similar critic reward growth trends during training, only MicroCoder dataset produces significant test set improvements, demonstrating that training effectiveness on challenging problems translates more directly to generalization performance.
- **Response length dynamics.** Harder problems exhibit accelerated response length growth with greater final magnitudes; MicroCoder dataset demonstrates faster growth rates and ultimately achieves longer outputs despite initially producing similar or shorter response lengths compared to DeepCoder dataset.

D.5 Context Length and Extension

- **Scaling relationship.** Longer maximum output lengths correlate with higher final accuracy, demonstrating clear scaling trends with model performance.
- **Growth dynamics.** Larger maximum output lengths drive faster output growth rates and greater final output lengths.
- **Diversity correlation.** Increased output or maximum output lengths positively correlate with higher output diversity.
- **Persistent limitation effects.** Initial use of small maximum output lengths reduces both response length and diversity, creating persistent performance deficits even after subsequent length extensions.
- **Limitation severity.** Smaller initial maximum output lengths produce greater negative impacts on response length and performance.
- **Irreversible training effects.** Extended training under small initial maximum output lengths amplifies negative effects on output diversity, response length, and model performance, with models showing minimal recovery when limitations are relaxed beyond specific training thresholds, indicating early-stage output reduction fundamentally alters learning trajectories.

D.6 Truncation Mask

- **Truncation masking mechanism.** Responses reaching maximum response length are excluded from training by setting advantage scores to zero, preventing truncated outputs from contributing to policy optimization.
- **Conditional truncation masking criteria.** Selectively masks responses that simultaneously reach maximum length, produce non-incorrect answers, avoid repetition sequences (final 128 tokens differ from preceding 128 tokens), and masks only a specified proportion rather than all qualifying responses.
- **Performance trajectory.** Masking creates distinct dynamics where training rapidly rises to higher values, then declines, and converges to specific performance levels.
- **Length growth acceleration.** Higher masking rates accelerate output length growth and push convergence values closer to maximum response limits, with 30% masking achieving growth rates comparable to complete masking.
- **Peak achievement speed.** Increased masking enables faster achievement of initial performance peaks.
- **Peak performance tradeoff.** Reduced masking extends the initial improvement phase and achieves higher peak performance values.
- **Diversity dynamics.** Increased masking accelerates response diversity decline and reduces diversity convergence values, with complete masking showing brief diversity increase followed by rapid descent.
- **Stability advantage.** Conditional truncation masking demonstrates superior training stability compared to both no masking and complete masking approaches, achieving significantly higher final performance while avoiding the rapid training collapse observed with complete masking.

D.7 Batch Size and On-Policy

- **Training configuration mechanism.** train_batch_size defines the number of problems for single inference round, ppo_mini_batch_size defines the number of problems used for single parameter update; model first performs inference on train_batch_size problems, obtains rewards for these responses, then updates parameters train_batch_size/ppo_mini_batch_size times using ppo_mini_batch_size problems per update, and proceeds to inference on next train_batch_size problems after completing all updates, and repeats this cycle iteratively throughout training.
- **On-policy versus off-policy spectrum.** Smaller train_batch_size values (maintaining constant ppo_mini_batch_size) create more on-policy behavior resembling immediate problem-solving reflection, while larger values produce off-policy dynamics akin to batch reflection after completing all problems.
- **Stability characteristics.** On-policy configurations exhibit reduced training stability with accelerated response diversity convergence and response length trends that rise then decline, whereas off-policy approaches demonstrate greater training stability.
- **Optimal performance balance.** Intermediate configurations balancing on-policy and off-policy characteristics achieve superior performance, outperforming heavily skewed settings in either direction.

D.8 KL Loss and Clip Ratio

- **Removal benefits.** Eliminating KL loss with high clipping enhances output diversity and response length, driving sustained performance improvements; standard KL loss without high clipping reduces output diversity and limits response length to marginal increases, causing modest initial performance gains followed by decline.
- **Diversity-performance relationship.** Continued diversity reduction creates unsustainable training dynamics where performance first rises then falls, preventing effective long-term training and model optimization.

E PROMPTS

E.1 DATA PROCESSING PROMPT

The following prompt is used in the Process stage of Appendix F.1 to clean, translate, and filter crawled problems.

```
# Programming Problem Quality Optimization Processing Instructions

You are a professional programming problem processing expert with
multilingual translation capabilities and web content cleaning skills.
Your task is to analyze and optimize programming problems crawled from
the web, clean up garbage information, perform necessary language
translation, and maintain the integrity and accuracy of the original
content to make it suitable for algorithm practice.

## Input Content
You will receive a programming problem crawled from the web, which may
contain various format and quality issues, and may be in non-English
content.

## Processing Principles

### 1. Quality Assessment

**High-quality problems (perfect=true)**:
- No modification needed
- May have the following characteristics:
- Clear and complete problem description with coherent logic
- Clear input and output formats
- Complete and correct examples
- Clear constraints
- In English
- Does not contain useless garbage information brought by crawlers

**Problems to abandon (abandon=true)**:
- Not worth modifying
- Only abandon when the problem is completely unprocessable, do not give
up easily
- May have the following characteristics:
- Missing key information that makes the problem unsolvable
- Seriously incomplete problem description or chaotic logic
- Missing images that cause the problem to be incomprehensible
- Seriously missing mathematical formulas that cannot be inferred
- Too much garbage information that cannot be cleaned
- Contains unsafe information

### 2. Specific Processing Requirements

**Language Processing**:
- Translate all non-English content (Chinese, Japanese, etc.) into clear
English
- Maintain accuracy of technical terminology
- Ensure content remains the same before and after translation, do not
subjectively modify or delete information
- Ensure logical integrity after translation

**Format Preservation**:
- Maintain the structure of the original problem
- Preserve the original structural hierarchy
- Maintain the format of code blocks and examples
- Maintain the representation of data structures

**Image Processing**:
```

```
- If images are crucial for understanding the problem but are missing,
mark as abandon
- If images are merely decorative, they can be ignored

**Mathematical Formula Processing**:
- Do not subjectively add, modify, or delete content
- Fix incomplete mathematical symbols and formulas
- Ensure completeness of mathematical expressions

**Garbage Information Cleaning**:
- Remove web navigation elements (buttons, links, etc.)
- Remove irrelevant advertising text
- Remove links unrelated to problem solving
- Clean up duplicate or useless tags
- Preserve core problem content

**Table and Chart Processing**:
- Ensure readability of table data
- Do not subjectively add, delete, or modify content

**Content Preservation Principles**:
- Strictly preserve original problem content and format
- Do not subjectively add, delete, or modify information
- Do not change problem structure
- Only perform necessary translation and garbage cleaning

## Output Requirements

Strictly output in the following JSON format, do not add any other
content:

{
    "abandon": true/false,
    "perfect": true/false,
    "question": "(Complete optimized problem content)",
    "note": "(Brief description of main modifications made)"
}

## Question:

Now please process the following programming question:

<question>
</question>
```

E.2 Difficulty Scoring Prompt

The following descriptors define the five dimensions of the difficulty matrix in Appendix F.2.2; each dimension is scored from 1 to 5 and combined with the weights given in Figure 11.

```
### **Problem Comprehension Difficulty**
- **1 Point - Straightforward**: Clear problem description, requirements
are obvious at a glance, no ambiguity
- **2 Points - Slightly Thoughtful**: Requires careful reading, 1-2
details need attention
- **3 Points - Requires Analysis**: Need to understand examples, derive
implicit conditions, abstract the model
- **4 Points - Relatively Complex**: Lengthy problem description,
multiple constraints need to be integrated and understood
- **5 Points - Highly Abstract**: Requires deep analysis to understand
the real requirements, may need mathematical modeling

### **Knowledge Breadth Requirements**
```

```
- **1 Point - Basic Syntax**: Variables, loops, conditional statements,
arrays, functions, basic input/output, etc.
- **2 Points - Basic Data Structures**: String operations, stack, queue,
linked list, simple hashing, set, dictionary, 2D arrays, etc.
- **3 Points - Common Data Structures**: Binary tree, graph, heap,
priority queue, union-find, prefix sum, difference array, monotonic stack,
 sliding window, etc.
- **4 Points - Advanced Knowledge**: Segment tree, binary indexed tree,
trie, suffix array, KMP algorithm, minimum spanning tree, shortest path,
network flow, mathematical formulas, probability and statistics, bit
manipulation techniques, state machines, etc.
- **5 Points - Specialized Domain**: Computational geometry, number
theory (Euler's function, Chinese Remainder Theorem), game theory, FFT,
polynomial algorithms, persistent data structures, balanced trees, AC
automaton, domain-specific algorithms, etc.

### **Algorithmic Thinking Complexity**
- **1 Point - Simulation Implementation**: Direct simulation by problem
statement, brute force enumeration, simple traversal, counting, finding
maximum/minimum values, etc.
- **2 Points - Simple Techniques**: Two pointers, fast-slow pointers,
greedy choices, simple recursion, prefix sum applications, sorting then
processing, hash table deduplication, etc.
- **3 Points - Classic Algorithms**: Binary search and its variants, BFS/
DFS, backtracking, basic dynamic programming, topological sorting, union-
find applications, divide and conquer, sliding window, etc.
- **4 Points - Complex Algorithms**: State compression DP, interval DP,
tree DP, digit DP, shortest path variants, bipartite graph matching,
monotonic queue optimization, memoization, complex greedy strategies, etc.

- **5 Points - Advanced Algorithms**: Multiple algorithm combinations, DP
 optimizations (slope optimization, quadrangle inequality), heuristic
search, solutions requiring mathematical proof, innovative algorithm
design, approximation algorithms for NP problems, etc.

### **Implementation Difficulty**
- **1 Point - Simple Implementation**: < 20 lines of code, no special
edge cases
- **2 Points - Standard Implementation**: 20-50 lines, a few edge cases
to handle
- **3 Points - Moderately Complex**: 50-100 lines, multiple functions
working together, careful detail handling required
- **4 Points - Complex Implementation**: > 100 lines, complex state
management, prone to bugs
- **5 Points - Extremely Challenging**: Extensive detail handling,
difficult to debug, requires sophisticated code organization

### **Optimization Depth**
- **1 Point - No Optimization Needed**: Basic solution is already optimal,
 O(n) or O(nlogn)
- **2 Points - Simple Optimization**: Obvious repeated computations can
be eliminated, space can be compressed
- **3 Points - Standard Optimization**: Time complexity can be reduced by
 one level (e.g., O(n^2) -> O(nlogn))
- **4 Points - Deep Optimization**: Requires algorithm replacement or
sophisticated space optimization
- **5 Points - Ultimate Optimization**: Requires combination of multiple
techniques, involves complex time-space tradeoffs
```

F MICROCODER-DATASET

This appendix documents the construction and evaluation of MicroCoder-Dataset.

F.1 Data Processing Framework

F.1.1 Overview

Our data processing framework (Figure 10) adopts a four-stage pipeline designed to construct high-quality coding datasets. In the Collect stage, data is gathered from a wide range of sources, including publicly available datasets (Taco, KodCode, DeepCoder, etc.) along with programming problems crawled from the web, in order to maximize topical coverage. The Process stage performs data standardization through language translation, noise removal, format normalization, and completeness validation. The Filter stage then enforces multi-criteria selection that takes into account textual quality, content relevance, and difficulty distribution. Lastly, the Verify stage carries out manual inspection to confirm that problems are readable, complete, and that the test cases are correct. Through this end-to-end pipeline, raw datasets are converted into a high-quality, standardized corpus well suited for reinforcement learning training.

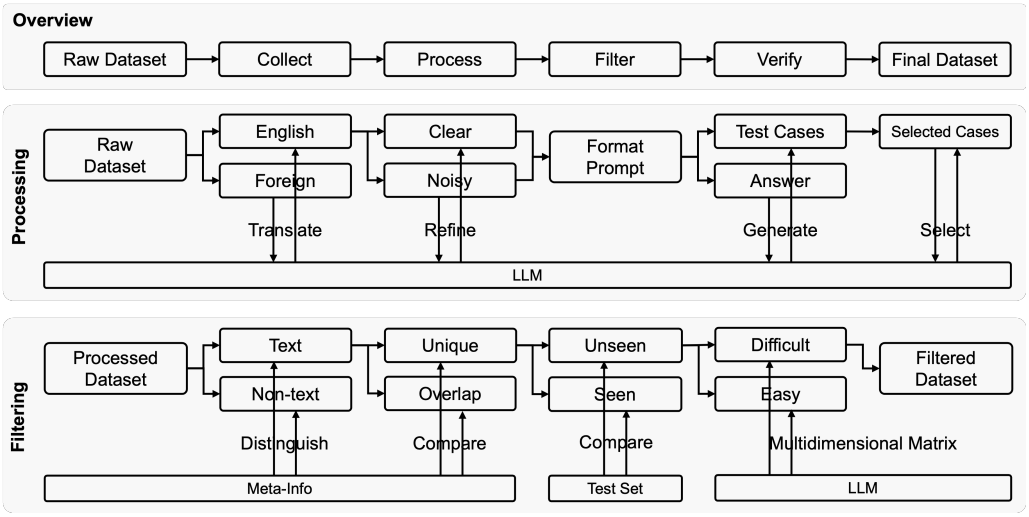


Figure 10: **Data Processing Framework Overview.** *Top:* A four-stage pipeline that converts raw datasets into high-quality training data via data collection, standardization, quality filtering, correctness verification, and final dataset construction. *Middle:* Standardization steps, including translation of non-English problems into English, noise removal and cleanup, unification of format prompts, and LLM-driven test case generation and selection. *Bottom:* Multi-stage filtering that enforces hard requirements (text-only problems through content-type discrimination, uniqueness through overlap detection, and unseen status through test set comparison) together with adaptive selection (difficulty-based filtering using a multidimensional difficulty matrix in which an LLM separates easy from difficult problems), producing the filtered dataset that is passed to the verification stage.

F.1.2 Processing

The processing stage carries out standardization through five core steps. Translation converts non-English problems (e.g., Japanese AIZU problems) into English so that they can be uniformly accessed. Noise removal targets a number of data quality issues: missing images that hinder problem comprehension, incomplete mathematical formulas and symbols, broken tables or figures, irrelevant crawled content such as hyperlinks and advertisements, truncated problem descriptions, and various content quality concerns. Problems that cannot be properly cleaned are discarded automatically, while the original problem accuracy is preserved without any subjective edits. Test case optimization addresses two main challenges. For problems that either lack test cases but include reference solutions, or whose test cases are noisy due to web scraping, an LLM is used to synthesize comprehensive test cases. Although LLMs cannot solve every problem, they are highly effective at producing test case inputs and reasoning about boundary conditions. By running the reference solution against these inputs, the system obtains outputs that are guaranteed to be correct, which makes it possible to

automatically generate comprehensive, accurate, and appropriately sized test suites. For problems that include excessively many test cases (hundreds per problem, leading to datasets exceeding 100GB across thousands of problems), the framework keeps the 15 longest test cases on the assumption that length is correlated with difficulty. Such large data volume affects processing, loading, training speed, and stability. The system also discards problems that have no test cases as well as those that require functional validation in which multiple correct answer formats are admissible. The processing stage therefore inherently performs filtering at the level of both problem quality and test case selection.

Format standardization unifies prompt structures: CF retains its native format, whereas the remaining sources are aligned to the LiveCodeBench format. This handles the differences in execution logic between LeetCode-style function completion problems and OJ-style input/output problems. Because the two problem types are mixed during model training, many of the original datasets do not provide explicit format instructions, which can cause the model to solve a problem correctly but emit code in an incorrect format.

F.1.3 FILTERING

The filtering stage applies multi-level criteria via two complementary mechanisms: hard requirements and adaptive selection. Hard requirements demand text-only problems, uniqueness ensured through overlap detection, and unseen status with respect to the test sets. Adaptive requirements build on the quality improvements from the processing stage and perform difficulty-based selection that is tailored to specific datasets and model capabilities, relying on a multidimensional, LLM-powered difficulty matrix. A detailed description of this matrix and how it is applied is provided in Section F.2.2 and F.2.3.

For train-test separation, we adopt a 16-gram similarity analysis with a threshold of 0.22. A validation experiment using AtCoder problems (the most recent training data that share sources with LiveCodeBench) against the LiveCodeBench v6 test set indicates that around 3% of training samples exceed the 0.22 similarity threshold, but none of them are identical to any test set problem. This shows that the 16-gram, 0.22 criterion is both comprehensive and efficient, and it is therefore applied uniformly across all datasets to guarantee train-test separation.

F.2 AUTOMATIC DIFFICULTY FILTERING

F.2.1 PREDICT-CALIBRATE-SELECT

Automatic Difficulty Filtering follows a three-stage predict-calibrate-select framework. In the Predict stage, an LLM is used to assess problem complexity through a multidimensional difficulty matrix, taking the average of three independent assessments per problem. The Calibrate stage links the predicted difficulty scores to actual model performance in order to fix difficulty category boundaries; here ground-truth difficulty is computed as the success rate of four model attempts on each problem. The Select stage then uses the calibrated boundaries to filter out overly easy problems, keeping only those that fall within the target difficulty range appropriate for the training objective.

F.2.2 MULTI-DIMENSIONAL DIFFICULTY METRICS

The Multi-dimensional Difficulty Metrics (Figure 11) decompose problem complexity along five weighted dimensions. Both the matrix itself and its associated weights are designed by drawing on theories from cognition, evaluation, and software engineering, including Bloom's Taxonomy of Educational Objectives (Bloom, 1956), General Evaluation Dimensions (Zhou et al., 2025), McCabe Complexity Theory (McCabe, 1976), and Halstead Complexity Measures (Halstead, 1977). Problem Comprehension Difficulty and Knowledge Breadth Requirements are assigned the smallest weights, since they primarily measure semantic understanding and memory. Algorithmic Thinking Complexity and Implementation Difficulty receive substantially higher weights because they probe reasoning and programming ability. Each dimension is rated on a 1-5 scale together with concrete descriptors, which makes evaluation consistent across problems. As an illustration, GPT-4O scores a grid manipulation problem in three independent assessments, yielding an average difficulty of 3.03, which places the problem in the top 30% of the dataset and marks it as relatively challenging. The validity of the matrix is supported by the performance gains reported in Table 1.

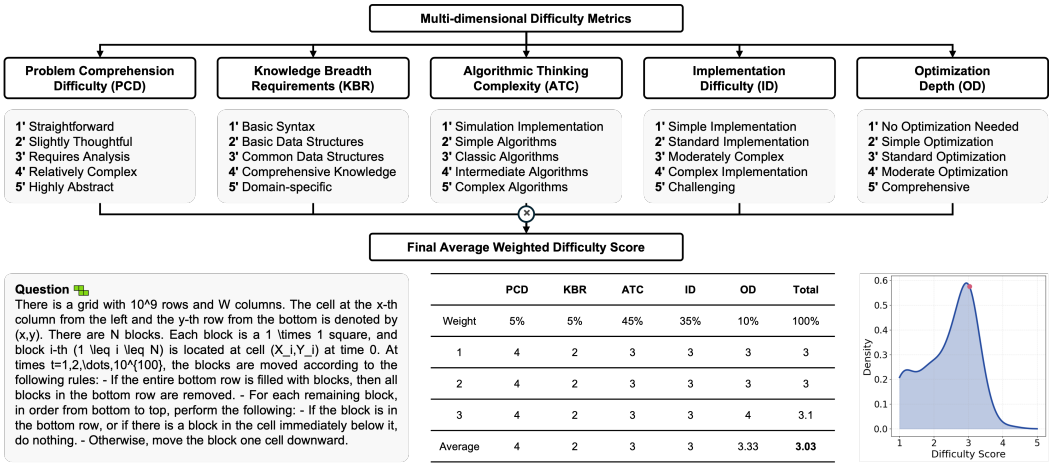


Figure 11: **Multi-dimensional Difficulty Metrics Framework.** *Top:* A five-dimensional assessment scheme in which each dimension is rated on a 1-5 scale with explicit descriptors that define the evaluation criteria. *Bottom Left:* An example of scoring a problem with three independent GPT-4O assessments. Each assessment rates all five dimensions (e.g., Assessment 1: PCD=4, KBR=2, ATC=3, ID=3, OD=3) and applies the dimension weights (ATC 45%, ID 35%, OD 10%, PCD 5%, KBR 5%) to produce per-assessment scores (3, 3, 3.1), which are averaged into a final difficulty score of 3.03. *Bottom Right:* A distribution curve showing that the example problem's difficulty score (3.03) places it in the top 30% of the dataset, marking it as a relatively challenging problem.

F.2.3 Case Study

The Case Study illustrates the full predict-calibrate-select pipeline using LiveCodeBench (Figure 12). First, GPT-4O scores problems with the multi-dimensional matrix, averaging three assessments per problem. Next, a subset of problems is empirically validated, in which Qwen-3-4B-thinking attempts each problem four times so that ground-truth difficulty can be derived from the success rate. Calibration identifies optimal difficulty boundaries at 2.5 and 2.75 for separating easy, medium, and hard categories, producing nearly identical predicted and empirical difficulty distributions. Finally, the filtering step discards every problem that scores below 2.5, removing 30% of the dataset overall while eliminating more than 65% of the easy problems and retaining the difficult ones. As a result, the proportion of easy problems is reduced from roughly 40% to under 20% in the final dataset.

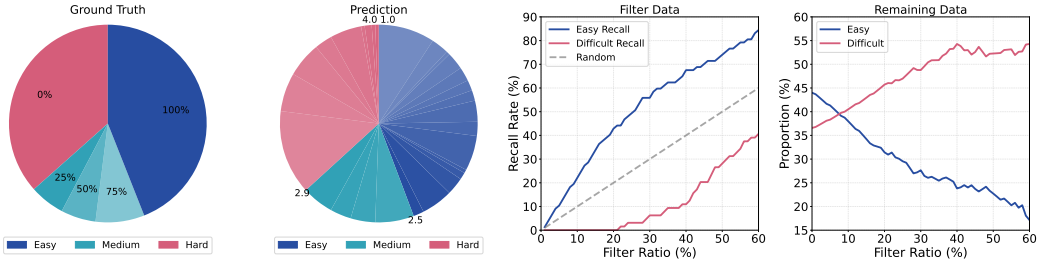


Figure 12: **Difficulty Filtering Case Study on LiveCodeBench v6.** *Left:* The empirical difficulty distribution measured from model pass rates. *Middle Left:* GPT-4O predicted difficulty calibrated against the ground truth to set boundaries at 2.5 (easy/medium) and 2.75 (medium/hard), achieving near-perfect alignment with the empirical distribution and supporting accurate difficulty-based filtering on subsequent data. *Middle Right:* Recall curves indicating the proportion of each difficulty category that is removed; high easy recall (over 60% at 30% filtering) together with low difficult recall demonstrates the selective removal of simple problems. *Right:* The difficulty composition of the retained problems; filtering 30% of the data shifts the distribution from 45% easy to under 25% easy, while pushing the proportion of difficult problems above 50%, which confirms that difficulty-based selection is effective.

F.3 Composition and Statistics

Dataset Composition. The MicroCoder dataset is built exclusively from real competitive programming problems and contains no synthetic data (Figure 13). The open-source portion is mainly drawn from the TACO and OlympicCoder datasets, while the privately collected portion gathers competition problems from a wide range of programming contest platforms. After strict quality and difficulty filtering, the initial corpus is reduced to 13,300 curated problems, with the largest contributions coming from the AIZU, AtCoder, CodeChef, and Kattis platforms.

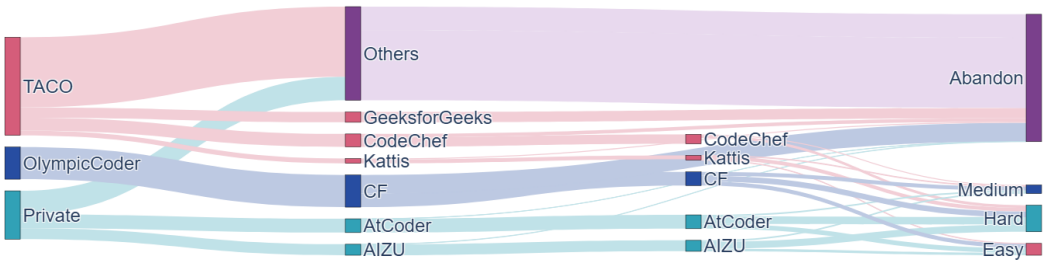


Figure 13: **Dataset Composition and Processing Flow.** A Sankey diagram showing the full data pipeline from heterogeneous sources to the final difficulty-based categorization. The dataset is composed solely of real competitive programming problems, with no generated content. Open-source contributions are subjected to heavy filtering, in which the majority of problems are dropped and primarily difficult ones retained. Private collections supply the bulk of the challenging and recent problems across platforms, yielding a high-quality and challenging dataset.

Dataset Statistics. The MicroCoder dataset displays a variety of statistical properties across platforms, as shown in Figure 14. The problem length distributions are fairly consistent across sources, with most problems concentrated between 1K and 3K characters, and AIZU problems exhibit the largest average length. Test case distributions reveal a clear advantage of the privately collected datasets, which provide more test cases per problem than the open-source datasets. To prevent training stalls caused by problems with hundreds of long test cases, we cap every dataset at the 15 longest test cases, following the approach of DeepCoder. The t-SNE clustering analysis reveals clear separation between platforms, confirming that privately collected and open-source datasets cover complementary content rather than redundant material. In addition, cosine similarity analysis between training datasets and test benchmarks (AtCoder, LeetCode, LiveCodeBench) yields uniformly low similarity scores ranging from 0.04 to 0.14, validating strict train-test separation with zero overlap and ensuring reliable evaluation.

F.4 Experimental Design

Benchmarks. We evaluate dataset quality on competitive programming benchmarks, including AtCoder and LeetCode from LiveCodeBench v6 (Jain et al., 2025), which cover strictly unseen problems released after the model training cutoff. These benchmarks span multiple difficulty levels (easy, medium, hard) and offer a comprehensive assessment of model performance on a diverse set of algorithmic challenges. Evaluation is performed using the official LiveCodeBench v6 testing framework and code execution infrastructure, which provides standardized and reproducible results.

Evaluation. Model performance is measured in terms of accuracy: each problem receives a binary score, where 1 indicates that the generated solution passes all test cases and 0 indicates that it does not. We run four independent inference attempts per problem and report the average accuracy across these attempts. Results are presented per difficulty category as well as for the overall benchmarks.

Baselines. We compare MicroCoder against the DeepCoder dataset (Luo et al., 2025), a widely used and open-source competitive programming dataset. To enable a fair comparison, both datasets are evaluated under identical training and inference configurations.

Models. The experiments use Qwen3-4B-Instruct-2507 (Yang et al., 2025), trained from its official checkpoint.

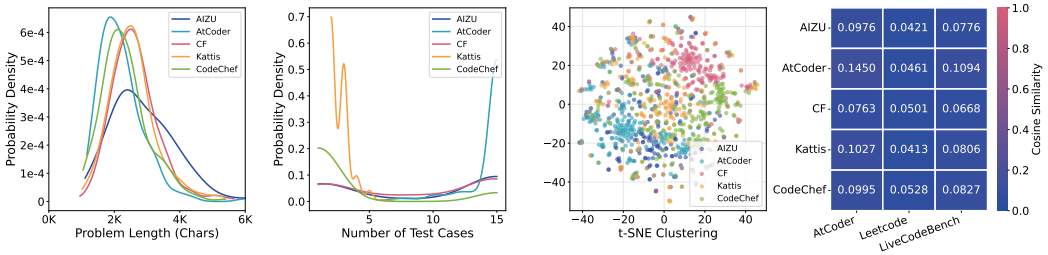


Figure 14: **Dataset Statistical Characteristics and Diversity Analysis.** *Left:* Problem length distribution across platforms, where the privately collected datasets (AIZU, AtCoder) display length profiles comparable to those of the open-source datasets (Kattis, CodeChef), with AIZU showing the longest average problem length. *Middle Left:* Test case distribution, where the privately collected datasets contain more test cases per problem; all datasets are capped at the 15 longest test cases following DeepCoder, in order to avoid training stalls caused by problems with hundreds of long test cases. *Middle Right:* A t-SNE clustering visualization that demonstrates clear platform separation and complementary coverage between privately collected and open-source datasets. *Right:* A cosine similarity table in which training datasets (y-axis) show low similarity (0.04-0.14) against the test benchmarks (x-axis: AtCoder, LeetCode, LiveCodeBench), confirming the absence of overlap between training and test sets.

Algorithms. For reinforcement learning training, we adopt GRPO and DAPO (Yu et al., 2025). DAPO drops the KL loss and uses high clipping to encourage the model to generate more diverse solutions.

Hyperparameters. Training with the GRPO and DAPO algorithms uses the following configuration: a maximum response length of 8K tokens, a temperature of 1.2, a training batch size of 64, a learning rate of 1e-6, 8 samples per query, and a binary 0-1 accuracy as the reward.

F.5 Results

F.5.1 Main Results Analysis

Training Dynamics. Figure 15 highlights the consistent advantages of MicroCoder under both the GRPO and DAPO algorithms. MicroCoder reaches higher test accuracy throughout training while attaining lower critic rewards on the training set, which indicates greater problem difficulty that nevertheless leads to stronger model generalization. This gap between training rewards and test performance supports our hypothesis that challenging problems, although harder to solve during training, lead to more effective capability gains. Compared with GRPO, DAPO produces higher test-set performance and more stable training dynamics, which we attribute to its removal of the KL loss and its use of high clipping to encourage solution diversity, leading to larger gains on challenging problems.

Overall Performance. Table 1 reports comprehensive benchmark results comparing the DeepCoder and MicroCoder datasets under identical training configurations. MicroCoder consistently outperforms DeepCoder across all benchmarks and difficulty levels. Under GRPO training, MicroCoder achieves absolute improvements of +2.2 points on AtCoder, +1.6 points on LeetCode, and +2.0 points on LiveCodeBench overall accuracy. The advantages become more pronounced under DAPO, where MicroCoder delivers over DeepCoder +3.6 points on AtCoder, +6.0 points on LeetCode, and +4.4 points on LiveCodeBench, corresponding to relative improvements of 9.3%, 18.7%, and 12.1%, respectively.

Difficulty-Specific Analysis. The performance gains exhibit clear difficulty-dependent patterns, which validates the effectiveness of difficulty-based filtering. On easy problems, the improvements are relatively modest because the two datasets are approaching performance convergence. Medium-difficulty problems show substantial gains, with MicroCoder achieving between +8.6 and +9.7 points over DeepCoder under DAPO across benchmarks, including a notable +40.4% relative improvement on LeetCode Medium. Hard problems show the largest relative improvements, with gains of +20.7%

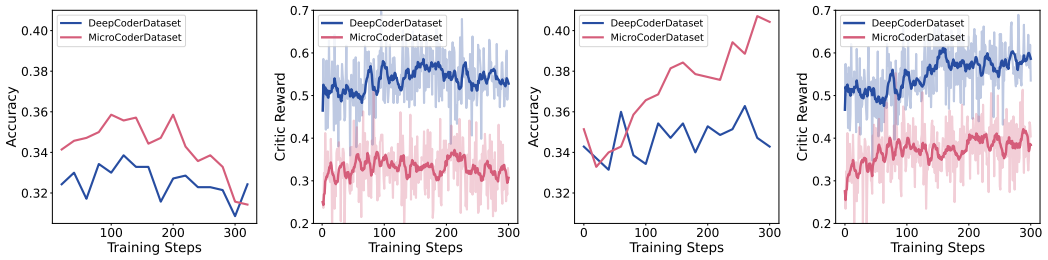


Figure 15: **Training Dynamics Comparison Between DeepCoder and MicroCoder Datasets.** Training curves under the GRPO (left) and DAPO (right) algorithms, with accuracy reported on the LiveCodeBench v6 test set and critic reward reported on the training set. Darker lines correspond to smoothed curves over the lighter raw training data. MicroCoder consistently attains higher test accuracy, indicating stronger capability gains for code generation models. DAPO, which removes the KL loss and applies high clipping to encourage diverse solutions, achieves better performance and more stable training than GRPO. MicroCoder yields lower training rewards than DeepCoder, indicating higher difficulty, yet delivers larger test improvements at comparable training progress, which validates the effectiveness of challenging problems for model generalization.

to +22.0% on LiveCodeBench Hard and AtCoder Hard problems under DAPO. These results confirm that the performance gains arise primarily from medium and hard problems, where model capabilities are stretched the most, and demonstrate that selecting challenging problems leads to performance gains across all difficulty levels.

F.5.2 Ablation Studies

To analyze the variables that influence performance and to identify the sources of performance gains, we run experiments across different model sizes and dataset sources using GRPO and its variants. The results show that MicroCoder consistently surpasses alternative sources at both scales. On 1.7B models trained with DAPO, MicroCoder reaches 23.2% on AtCoder and 22.0% on LiveCodeBench, considerably higher than OlympicCoder (22.5%, 19.6%) and DeepCoder (22.3%, 19.7%). When scaling up to 4B models, MicroCoder reaches 42.2% on AtCoder and 40.7% on LiveCodeBench, exhibiting larger absolute gains over the baselines than at smaller scale.

These results suggest several insights. First, MicroCoder's advantages become more pronounced as model capabilities increase, which validates the claim that challenging and recent problems contribute more once models have enough capacity to take advantage of them. Second, MicroCoder's edge does not stem solely from filtering existing problems, but from incorporating recent and challenging problems that improve model capabilities. Third, DAPO consistently produces larger improvements than GRPO across all settings, which shows that diversity-driven training algorithms achieve greater gains on challenging problems. Together, these findings underscore the importance of data recency and difficulty when training state-of-the-art code generation models.

G A Complete MicroCoder-Dataset Record

The listing below is a representative released record from the pipeline in Appendix F.1. It is a collected AIZU problem, not a synthetic one, and keeps the platform, problem identifier, and source URL for traceability. The prompt carries the unified stdin and stdout instruction in the LiveCodeBench style, and the test suite covers all 46 admissible inputs, so the binary reward is a full correctness check.

```
{
  "data_source": "aizu",
  "prompt": [
    {
      "role": "user",
      "content": "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.\n\n#
```

```
Discounts of Buckwheat\n\nAizu is famous for its buckwheat. There are
many people who make buckwheat noodles by themselves.\n\nOne day, you
went shopping to buy buckwheat flour. You can visit three shops, A, B and
 C. The amount in a bag and its unit price for each shop is determined by
 the follows table. Note that it is discounted when you buy buckwheat
flour in several bags.\n\n|  | Shop A | Shop B | Shop C |\n| --- | --- |
--- | --- |\n| Amount in a bag | 200g | 300g | 500g |\n| Unit price for a
 bag (nominal cost) | 380 yen | 550 yen | 850 yen |\n| Discounted units |
 per 5 bags | per 4 bags | per 3 bags |\n| Discount rate | reduced by 20
% | reduced by 15 % | reduced by 12 % |\n\nFor example, when you buy 12
bags of flour at shop A, the price is reduced by 20 % for 10 bags, but
not for other 2 bags. So, the total amount shall be (380 x 10) x 0.8 +
380 x 2 = 3,800 yen.\n\nWrite a program which reads the amount of flour,
and prints the lowest cost to buy them. Note that you should buy the
flour of exactly the same amount as the given input.\n\n## Input\n\nThe
input consists of multiple datasets. For each dataset, an integer a (500
<= a <= 5000, a is divisible by 100) which represents the amount of flour
 is given in a line.\n\nThe input ends with a line including a zero. Your
 program should not process for the terminal symbol. The number of
datasets does not exceed 50.\n\n## Output\n\nFor each dataset, print an
integer which represents the lowest cost.\n\n## Sample Input\n\n```\n\
n500\n2200\n0\n\n```\n\n## Output for the Sample Input\n\n```\n850\
n3390\n\n```\n\n---\n\nSource: <https://onlinejudge.u-aizu.ac.jp/problems
/0106>\n\n### Format: 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 delimiters as follows. Ensure that when the
python program runs, it reads the inputs, runs the algorithm and writes
output to STDOUT.\n```python\n# YOUR CODE HERE\n```\n\n### Answer: (use
the provided format with backticks)"
    }
  ],
  "ability": "code",
  "reward_model": {
    "style": "rule",
    "ground_truth": "[{\"input\": \"500\\n600\\n700\\n800\\n900\\n1000\\
n1100\\n1200\\n1300\\n1400\\n1500\\n1600\\n1700\\n1800\\n1900\\n2000\\
n2100\\n2200\\n2300\\n2400\\n2500\\n2600\\n2700\\n2800\\n2900\\n3000\\
n3100\\n3200\\n3300\\n3400\\n3500\\n3600\\n3700\\n3800\\n3900\\n4000\\
n4100\\n4200\\n4300\\n4400\\n4500\\n4600\\n4700\\n4800\\n4900\\n5000\\n0
\", \"output\": \"850\\n1100\\n1230\\n1400\\n1610\\n1520\\n1950\\n1870\\
n2070\\n2250\\n2244\\n2620\\n2624\\n2794\\n3004\\n3040\\n3344\\n3390\\
n3590\\n3740\\n3764\\n4120\\n4114\\n4314\\n4494\\n4488\\n4864\\n4868\\
n5038\\n5248\\n5284\\n5588\\n5634\\n5834\\n5984\\n6008\\n6364\\n6358\\
n6558\\n6738\\n6732\\n7108\\n7112\\n7282\\n7492\\n7528\"}]"
  },
  "extra_info": {
    "split": "test",
    "index": 102,
    "reference": "...",
    "id": "aizu/0106",
    "website": "aizu"
  }
}
```


H EVALUATOR VALIDATION

Table 6 validates both evaluators against human labels on 500 model outputs on MicroCoder-Dataset problems, labeled by three annotators with majority vote, 253 correct and 247 incorrect. False-accept rates are over the 247 incorrect outputs and false-reject rates over the 253 correct outputs. The false rejects of the LiveCodeBench evaluator come from format variants of correct outputs, and its false accepts from harness edge cases such as crashed solutions whose empty output matches an empty

expected output, which MicroCoder-Evaluator rejects through execution-status and output-structure validation.

Table 6: **Validation against human labels on 500 outputs.** Rates with the corresponding counts.

Metric	MicroCoder-Evaluator		LiveCodeBench Evaluator	
	Rate	Count	Rate	Count
Agreement with human labels	98.6%	493/500	89.2%	446/500
False-accept rate	0.4%	1/247	1.2%	3/247
False-reject rate	2.4%	6/253	20.2%	51/253

Table 7 reports false-accept rates on 1,000 near-miss outputs, 200 per category, that match the expected format but are semantically wrong. Neither evaluator accepts any of them, since every rule of MicroCoder-Evaluator normalizes format and never rescues a wrong value.

Table 7: **False-accept rates on 1,000 near-miss outputs.** Each category contains 200 outputs.

Category	MicroCoder-Evaluator	LiveCodeBench Evaluator
Off-by-one numeric values	0.0%	0.0%
Permuted order of order-sensitive answers	0.0%	0.0%
Truncated sequences	0.0%	0.0%
Values just beyond float tolerance	0.0%	0.0%
Wrong values under type coercion	0.0%	0.0%
Overall	0.0%	0.0%

I Worked Examples of Evaluator Behavior

Table 8 lists eight output variants that preserve the content of a correct answer while changing only its rendering, such as spacing, numeric notation, container type, or floating-point rounding. The LiveCodeBench evaluator rejects every one of them and MicroCoder-Evaluator accepts every one of them, and each row names the rule involved, which can be checked against the released evaluator code. All rules act on rendering and never on values, so they cannot rescue a wrong answer, consistent with the near-miss results in Table 7.

Table 8: **Worked examples of evaluator behavior.** Eight semantics-preserving output variants that the LiveCodeBench evaluator rejects and MicroCoder-Evaluator accepts, with the rule involved.

Case	Model output	Ground truth	LiveCodeBench evaluator	MicroCoder-Evaluator
Floating-point precision	`3.14159265358979`	`3.14159265358980`	**Wrong**, exact decimal comparison finds inequality	**Correct**, approximate numeric comparison within tolerance
Extra whitespace between elements	`1  2  3` `4  5  6`	`1 2 3` `4 5 6`	**Wrong**, string comparison fails after stripping	**Correct**, splits by whitespace and compares token lists
Trailing whitespace on lines	`10 20 30` `40 50 60`	`10 20 30` `40 50 60`	**Wrong**, per-line comparison fails	**Correct**, strips each element after splitting

Empty lines in output	`42`
	
	`100`
	
	`200`
		`42`
		`100`
		`200`
			Wrong, line count mismatch of 5 versus 3	**Correct**, filters empty lines before comparison

Nested tuple versus list	`[(1, 2), (3, 4), (5, 6)]`	`[[1, 2], [3, 4], [5, 6]]`	**Wrong**, only the top-level tuple is converted	**Correct**, recursive conversion of tuples to lists
Float versus integer rendering	`1.0 2.0 3.0` `4.0 5.0 6.0`	`1 2 3` `4 5 6`	**Wrong**, `1.0` and `1` differ as strings	**Correct**, compares the values as floats
Scientific versus decimal notation	`1.5e+02` `2.0e+03`	`150` `2000`	**Wrong**, string comparison fails	**Correct**, compares the values as floats

Leading zeros	`007`
	`042`
	`100`
		`7`
		`42`
		`100`
			Wrong, `007` and `7` differ as strings	**Correct**, numeric conversion drops leading zeros