CorrGRPO: Correlation-Normalized GRPO for Multi-Reward Learning

Anonymous authors
Paper under double-blind review

Abstract

Group Relative Policy Optimization (GRPO) is widely used to train reasoning language models, where it computes advantages by centering and normalizing rewards across rollouts of the same prompt. For multiple rewards, GRPO sums the reward components and normalizes the total reward by its within-group standard deviation. The corresponding variance equals the sum of all pairwise reward covariances. For a fixed centered reward, larger aggregate covariance produces smaller advantages, and vice versa, allowing update magnitudes to adapt to reward dependence. However, correlated rewards with large scales can dominate this normalization and suppress signals from smaller-scale rewards. We propose **Corr**elation-Normalized **GRPO** (**CorrGRPO**), which normalizes pairwise covariances into Pearson correlation coefficients. CorrGRPO keeps the centered total reward unchanged while balancing the influence of differently scaled rewards on the correlation-based normalization. This allows advantage magnitudes to adapt to reward correlations without the normalization being dominated by large-scale reward components. We compare CorrGRPO with GRPO and other variants on code generation, tool calling, and agent security, using models ranging from 0.5B to 8B parameters. These tasks all involve multiple rewards that can improve together or present tradeoffs. Results show improvements across three domains, including code generation, tool calling, and agent security.

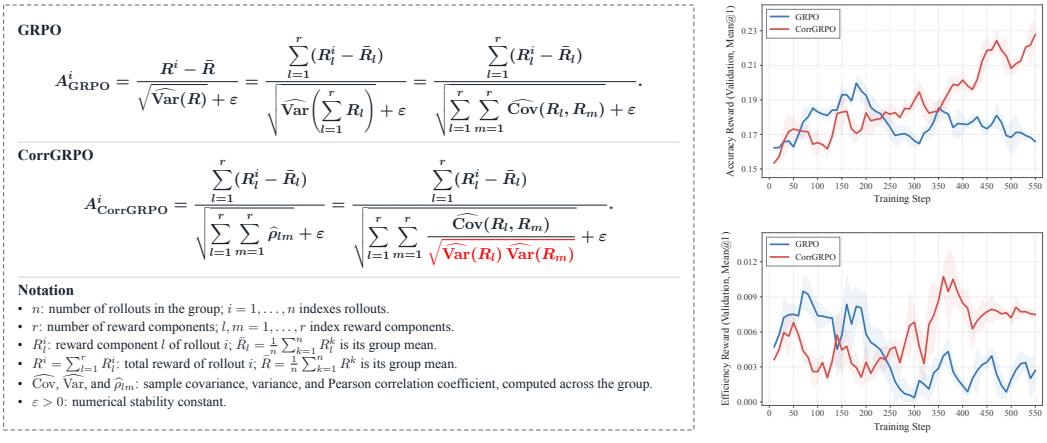


Figure 1: An overview of CorrGRPO. (1) Left: CorrGRPO replaces total-reward standard deviation normalization with a correlation-based denominator while retaining the centered total reward. (2) Right: Validation accuracy (top) and efficiency (bottom), measured by Mean@1, for Qwen2.5-Coder-7B-Instruct on LeetCodeDataset.

1 Introduction

Reinforcement learning (RL) has become a central paradigm for aligning large language models (LLMs) with human preferences and improving their ability to solve complex tasks.

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.

By constructing training environments and designing rewards that capture desired outcomes, RL enables models to improve through feedback on their own generations, supporting both instruction following and the development of reasoning capabilities (Ouyang et al., 2022; DeepSeek-AI et al., 2025). Among existing approaches, Group Relative Policy Optimization (GRPO) has gained widespread adoption for its simplicity and efficiency. GRPO estimates advantages by subtracting the mean reward and dividing by the reward standard deviation within a group of responses sampled for the same prompt, eliminating the need for a separate value model (Shao et al., 2024).

Practical training objectives often require multiple rewards to capture different aspects of desirable behavior. First, reward objectives can reinforce one another. Code generation, for example, can combine rewards for executability, test-case pass rate, and abstract syntax tree (AST) similarity to reference solutions; resolving syntax or runtime errors can improve both executability and test-case performance. Second, reward objectives can introduce tradeoffs. Agent training requires balancing task utility with security, where an overly conservative agent may avoid malicious instructions by also refusing legitimate requests, while an overly permissive agent may complete more tasks at the cost of greater exposure to prompt injection (Debenedetti et al., 2024). Such settings involve rewards that may improve together or exhibit tradeoffs. Their statistical relationships therefore provide a useful perspective on how multiple feedback signals interact during learning. In this work, we investigate how correlations among reward components affect advantage estimation in GRPO.

Our starting point is a simple identity with direct implications for multi-reward optimization. When GRPO aggregates r reward components into a total reward $R = \sum_{l=1}^{r} R_l$, the variance underlying its normalization is $\widehat{\mathrm{Var}}(\sum_{l=1}^{r} R_l) = \sum_{l=1}^{r} \sum_{m=1}^{r} \widehat{\mathrm{Cov}}(R_l, R_m)$. The underlying population identity is derived in Appendix L.1. Thus, the denominator depends on the sum of all entries in the reward covariance matrix, incorporating both individual reward variances and pairwise dependencies. Holding the marginal reward variances and centered total reward fixed, stronger positive correlations reduce the advantage magnitude by increasing the normalization denominator; weaker or more negative correlations increase the advantage magnitude by decreasing the denominator. GRPO therefore implicitly adjusts advantage scaling according to reward correlation within each rollout group. This provides a dynamic mechanism for modulating the strength of the aggregate learning signal as relationships among rewards change.

However, this mechanism entangles reward dependence with reward scale. Each covariance satisfies $\mathrm{Cov}(R_l, R_m) = \sigma_l \sigma_m \rho_{lm}$, where σ_l and σ_m are the component standard deviations and ρ_{lm} is their Pearson correlation coefficient. Consequently, reward components with large scales of variation can dominate both the diagonal and off-diagonal terms in the covariance sum. The normalization then becomes disproportionately sensitive to these components, while smaller-scale rewards contribute little to the adjustment. This imbalance suppresses the influence of smaller-scale rewards on normalization, even when they encode useful distinctions among responses. This limitation is particularly relevant when heterogeneous rewards differ substantially in their numerical ranges or within-group variability.

To address this issue, we propose Correlation-Normalized GRPO (CorrGRPO). As shown in Figure 1, CorrGRPO replaces pairwise covariances in the denominator with Pearson correlation coefficients while retaining the centered total reward in the numerator. Normalizing each covariance by the corresponding component standard deviations removes its dependence on reward scale. Each reward component with nonzero within-group variance consequently contributes equally to the diagonal of the correlation matrix, while off-diagonal terms reflect the strength and direction of pairwise dependence. The denominator remains responsive to reward correlations without being dominated by components solely because they vary on larger numerical scales. Meanwhile, preserving the numerator retains the relative reward weights specified by the training objective. As a result, CorrGRPO allows correlations involving smaller-scale rewards to influence advantage normalization without being downweighted by their scales.

We conduct experiments with CorrGRPO on code generation, tool calling, and agent security, three domains that naturally require multiple reward signals, using models ranging from 0.5B to 8B parameters. Comparisons with GRPO and relevant variants show improve-

ments in code accuracy and efficiency, tool-call accuracy, and the balance between agent utility and security. Together, these results support correlation-based normalization as an effective approach to improving practical multi-reward learning.

Our contributions are threefold:

1. **A covariance perspective on GRPO.** We characterize how the reward covariance matrix implicitly controls advantage scaling in multi-reward GRPO and show that large-scale rewards can dominate this adjustment.
2. **Correlation-normalized advantage estimation.** We introduce CorrGRPO, which replaces covariance-based normalization with correlation-based normalization while preserving the centered total reward, preventing large-scale rewards from disproportionately influencing the denominator.
3. **Experiments across three multi-reward domains.** We conduct extensive experiments on code generation, tool calling, and agent security across models from 0.5B to 8B parameters, demonstrating improvements over GRPO and relevant variants in the three domains, along with an expansion of the empirical Pareto frontier between reward objectives.

2 A Covariance Normalization View of Multi-Reward GRPO

Group Relative Policy Optimization (GRPO) (Shao et al., 2024) estimates advantages using rewards from a group of trajectories, avoiding a separate value model. For each prompt q, it samples n trajectories $\{\tau_i\}_{i=1}^n$ from an old policy $\pi_{\theta_{\text{old}}}$. With r reward components, let $R_l^i = R_l(q, \tau_i)$ and $R^i = \sum_l R_l^i$, with group means $\bar{R}_l = \frac{1}{n}\sum_i R_l^i$ and $\bar{R} = \frac{1}{n}\sum_i R^i$. Any fixed reward weights are absorbed into the corresponding components. Under outcome supervision, all generated tokens in trajectory i share the same advantage A^i_{GRPO}. The advantage is computed by centering the total reward and normalizing it by its within-group standard deviation: $A^i_{\text{GRPO}} = \frac{R^i - \bar{R}}{\sqrt{\widehat{\text{Var}}(R) + \varepsilon}}$, where $\varepsilon > 0$ ensures numerical stability.

Our observation is that, with multiple reward components, GRPO implicitly uses the aggregate reward covariance to control advantage scaling. Applying the variance-of-a-sum identity, as derived in Appendix L.1, we obtain

$$A^i_{\text{GRPO}} = \frac{R^i - \bar{R}}{\sqrt{\widehat{\text{Var}}(R) + \varepsilon}} = \frac{\sum_{l=1}^{r}(R_l^i - \bar{R}_l)}{\sqrt{\widehat{\text{Var}}\left(\sum_{l=1}^{r} R_l\right) + \varepsilon}} = \frac{\sum_{l=1}^{r}(R_l^i - \bar{R}_l)}{\sqrt{\sum_{l=1}^{r}\sum_{m=1}^{r}\widehat{\text{Cov}}(R_l, R_m) + \varepsilon}}. \tag{1}$$

Here, $\widehat{\text{Var}}$ and $\widehat{\text{Cov}}$ are computed across the same rollout group with a consistent normalization convention. This decomposition reveals an implicit mechanism in GRPO: the denominator adjusts advantage scaling through both individual reward variances and pairwise reward dependencies.

Specifically, let $\Delta R^i = R^i - \bar{R}$ and $S = \sum_{l,m}\widehat{\text{Cov}}(R_l, R_m)$. The decomposition gives $A^i_{\text{GRPO}} = g\Delta R^i$, where $g = (\sqrt{S} + \varepsilon)^{-1}$ is a scaling coefficient shared by all trajectories in the group. Positive covariances increase the variability of the summed reward, reducing this coefficient, while negative covariances offset part of that variability and increase it. Holding the marginal reward variances and a trajectory's centered total reward fixed, stronger positive correlations therefore reduce its advantage magnitude; weaker or more negative correlations increase it. Because these statistics are recomputed from newly sampled trajectories, the scaling coefficient adapts to reward relationships across prompts and throughout training. This mechanism changes the scale of the group's reward-driven contribution to the surrogate objective while preserving the relative reward weights in the numerator.

However, covariance couples statistical dependence with reward scale. Each pairwise correlation is weighted by the product of the component standard deviations: $S = \sum_{l=1}^{r}\hat{\sigma}_l^2 +$

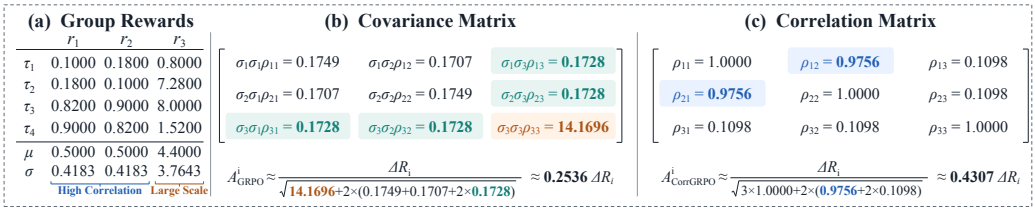


Figure 2: An example for comparing GRPO and CorrGRPO. **(a)** A group of four trajectories with three rewards. Rewards r_1 and r_2 are highly correlated, while r_3 has a larger scale and weak correlations with both. **(b)** Covariance-matrix elements. In GRPO, the large scale of r_3 dominates the denominator. **(c)** Pearson correlation coefficients matrix elements. In CorrGRPO, the strong correlation between r_1 and r_2 has a greater influence on the normalization denominator than r_3 does.

$2\sum_{l<m}\hat{\sigma}_l\hat{\sigma}_m\hat{\rho}_{lm}$, where $\hat{\sigma}_l^2 = \widehat{\mathrm{Var}}(R_l)$. Large-scale rewards can dominate both the variance terms and the sensitivity of the denominator to changes in correlation. Consequently, the shared scaling coefficient can be driven primarily by these components, while dependencies among smaller-scale rewards have limited influence on the adjustment.

Figure 2 illustrates this scale imbalance with four trajectories and three rewards. The first two rewards are strongly correlated ($\hat{\rho}_{12} \approx 0.9756$), while the third has weak correlations with both ($\hat{\rho}_{13} = \hat{\rho}_{23} \approx 0.1098$). Nevertheless, the third reward's variance, 14.1696, alone accounts for approximately 91.1% of the covariance sum $S = 15.5520$. Its covariances with the other rewards, both 0.1728, also exceed the covariance 0.1707 between the strongly correlated first two rewards. Thus, the third reward's large scale dominates the normalization despite its weak correlations, motivating CorrGRPO's removal of component-scale effects from the pairwise normalization terms.

3 CorrGRPO: Correlation-Normalized GRPO

Motivated by the scale imbalance identified in Section 2, we propose Correlation-Normalized GRPO (CorrGRPO), which replaces the covariance terms in GRPO's denominator with sample Pearson correlation coefficients:

$$A^i_{\mathrm{CorrGRPO}} = \frac{\sum_{l=1}^{r}(R_l^i - \bar{R}_l)}{\sqrt{\sum_{l=1}^{r}\sum_{m=1}^{r}\hat{\rho}_{lm}} + \varepsilon}, \qquad \hat{\rho}_{lm} = \frac{\widehat{\mathrm{Cov}}(R_l, R_m)}{\sqrt{\widehat{\mathrm{Var}}(R_l)\widehat{\mathrm{Var}}(R_m)}}. \tag{2}$$

All statistics are computed within the same rollout group, following Section 2. For zero-variance reward components, we set the corresponding rows and columns of the correlation matrix to zero, including diagonal entries, as shown in the core implementation in Appendix B. The resulting matrix remains positive semidefinite, ensuring a nonnegative correlation sum and a strictly positive denominator when $\varepsilon > 0$, with derivation in Appendix L.5.

Replacing covariances with correlations removes reward-scale weighting from each pairwise term. Each reward contributes one to the diagonal, while off-diagonal entries reflect the strength and direction of reward dependence. Consequently, strong correlations between small-scale rewards can substantially influence normalization even when other components have much larger variances. For example, in Figure 2, CorrGRPO assigns a correlation term of 0.9756 to the strongly correlated rewards r_1 and r_2, compared with 0.1098 for each weak relationship involving r_3. The stronger correlation contributes approximately 8.89 times as much to the sum inside the denominator.

4 Experiments

We evaluate whether CorrGRPO improves task performance when language models are trained with multiple reward signals. Our experiments cover code generation, tool calling, and agent utility and security. We compare CorrGRPO with GRPO (Shao et al., 2024)

Table 1: Coding RL results on LeetCodeDataset, HumanEval, MBPP, and LCB v6. Efficiency is the mean percentage of eligible programs that run faster than the reference code. Executable denotes running time success rate. Avg. is the arithmetic mean of the Pass@1 evaluation results across 4 datasets.

Model	LeetCodeDataset (In-dataset Eval)			HumanEval	MBPP	LCB v6	Avg.
	Efficiency	Executable	Pass@1	Pass@1	Pass@1	Pass@1	Pass@1
Qwen2.5-Coder-0.5B-Instruct	50.00	61.40	1.75	54.88	33.60	5.14	23.84
+ GRPO	60.71	87.72	3.07	57.32	40.00	3.43	25.96
+ GDPO	60.94	88.60	3.51	57.32	38.40	4.57	25.95
+ CorrGRPO	**67.86**	**89.04**	**3.95**	**59.76**	**42.20**	**6.29**	**28.05**
Qwen2.5-Coder-1.5B-Instruct	**58.93**	75.44	3.07	61.59	54.40	10.86	32.48
+ GRPO	40.00	84.21	6.58	64.63	58.00	**13.14**	35.59
+ GDPO	45.00	85.09	7.02	63.41	**58.60**	12.57	35.40
+ CorrGRPO	46.71	**86.84**	**8.77**	**68.90**	57.60	10.29	**36.39**
Qwen2.5-Coder-3B-Instruct	54.41	77.63	7.46	82.32	60.80	17.14	41.93
+ GRPO	56.00	**83.77**	12.28	80.49	63.80	16.57	43.29
+ GDPO	48.96	83.77	10.96	81.71	64.40	16.57	43.41
+ CorrGRPO	**60.00**	81.58	**14.47**	**82.32**	**66.00**	**19.43**	**45.56**
Qwen2.5-Coder-7B-Instruct	44.35	80.70	14.91	78.66	76.40	20.57	47.64
+ GRPO	44.70	**85.53**	15.79	73.78	78.40	21.14	47.28
+ GDPO	41.28	84.21	16.23	76.83	**79.60**	22.86	48.88
+ CorrGRPO	**51.42**	80.70	**24.12**	**78.66**	78.60	**24.57**	**51.49**

and the corresponding base models, and additionally include GDPO (Liu et al., 2026) in the coding experiments. We report both overall task performance and individual reward-related metrics to examine how CorrGRPO balances different objectives. Furthermore, we show our training dynamics in Figure 4 and provide detailed training settings in Appendix I.

4.1 Coding Reasoning

Task. We study coding RL with Python code generation. The goal is to produce functionally correct programs while improving execution efficiency. We evaluate generated programs using executable tests and compare the runtime of passing solutions with that of the reference implementations.

Reward design. Let y_g denote the generated program and y_r the reference implementation. We combine seven reward components:

$$\begin{aligned} R_{\text{code}}(y_g, y_r) = 0.05R_{\text{fmt}}(y_g) + 0.05R_{\text{syn}}(y_g) + 0.05R_{\text{compile}}(y_g) + 0.25R_{\text{run}}(y_g) \\ + 0.60R_{\text{pass}}(y_g) + 0.30R_{\text{ast}}(y_g, y_r) + 0.20R_{\text{eff}}(y_g, y_r). \end{aligned} \tag{3}$$

The format reward R_{fmt} checks whether the response follows the required code format. The execution-related rewards follow a dependency chain:

$$\underbrace{\text{Syntax validity}}_{R_{\text{syn}}} \rightarrow \underbrace{\text{Successful compilation}}_{R_{\text{compile}}} \rightarrow \underbrace{\text{Runtime success}}_{R_{\text{run}}} \rightarrow \underbrace{\text{All tests passed}}_{R_{\text{pass}}}.$$

These rewards are binary and dependent: the compilation reward requires valid syntax, the runtime reward requires successful compilation, and the all-pass reward requires execution without runtime errors. Improving earlier checks therefore enables rewards from subsequent checks. Given the set of test cases $\mathcal{C}$, the final correctness reward is $R_{\text{pass}}(y_g) = \prod_{c \in \mathcal{C}} \mathbf{1}[y_g \text{ passes } c]$. Thus, $R_{\text{pass}}(y_g) = 1$ only when every test passes, while syntax, compilation, and runtime rewards provide intermediate feedback even when $R_{\text{pass}}(y_g) = 0$. The structural reward $R_{\text{ast}}(y_g, y_r) \in [0, 1]$ measures the similarity between the generated and reference abstract syntax trees, independently of identifier names and literal values. Appendix J.1 describes the tree representation and similarity computation. Finally, the efficiency reward encourages faster execution among functionally correct programs: $R_{\text{eff}}(y_g, y_r) = \mathbb{I}[R_{\text{pass}}(y_g) = 1] \cdot \text{clip}\Big(1 - \frac{t(y_g)}{t(y_r)}, 0, 1\Big)$, where $t(y_g)$ is the execution

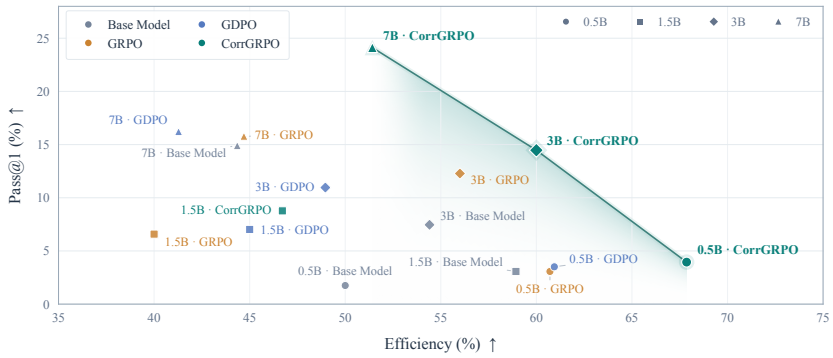


Figure 3: Pareto frontier of correctness–efficiency tradeoff on LeetCodeDataset.

time of the generated program and $t(y_r)$ is the mean execution time over four independent runs of the reference implementation. Both are measured using the same test harness.

Experimental setting. We use Qwen2.5-Coder-Instruct (Hui et al., 2024) at 0.5B, 1.5B, 3B, and 7B scales. Models are trained on the 2,641-problem training split of LeetCodeDataset (Xia et al., 2025) and evaluated on its 228-problem test split. This dataset contains programming problems paired with reference solutions and executable tests. Training hyperparameters are provided in Appendix I.1. To assess generalization without further training, we additionally evaluate HumanEval (Chen et al., 2021), which tests function completion from natural-language specifications; MBPP (Austin et al., 2021), which contains basic Python programming tasks; and LiveCodeBench v6 (Jain et al., 2024), which contains competition programming problems. Table 1 reports Executable, Pass@1, and Efficiency on LeetCodeDataset, together with Pass@1 on the three additional benchmarks. Executable measures the fraction of generated programs that execute without runtime errors, and Pass@1 measures the fraction that pass all tests. Efficiency measures the percentage of eligible generated programs that run strictly faster than their reference implementations, i.e., speedup $= t_{\text{ref}}/t_{\text{gen}} > 1$. A pair is eligible when both programs pass all tests and have positive, finite runtimes. We re-execute both programs in eight paired rounds on the same CPU core, alternating their order, and report the mean of the eight per-round percentages.

Main Results. CorrGRPO consistently improves coding correctness across model scales and extends the empirical correctness–efficiency Pareto frontier. Table 1 shows that CorrGRPO achieves the highest LeetCodeDataset Pass@1 and average benchmark Pass@1 at every evaluated model scale, outperforming both GRPO and GDPO. Relative to GRPO, average Pass@1 increases by 2.09, 0.80, 2.27, and 4.21 percentage points at 0.5B, 1.5B, 3B, and 7B, respectively. Figure 3 further shows that CorrGRPO accounts for every nondominated configuration across the evaluated methods and model scales. Its 7B model achieves the highest Pass@1 at 24.12%, its 0.5B model achieves the highest Efficiency at 67.86%, and its 3B model occupies an intermediate frontier point with 14.47% Pass@1 and 60.00% Efficiency.

4.2 Tool Calling

Task. We study structured tool calling, where a model selects the appropriate functions and generates their arguments from a user request and the available tool descriptions. Successful tool use requires jointly identifying the correct function, supplying the required parameter names, and assigning the correct parameter values. We evaluate both individual field correctness and complete-call correctness.

Reward design. Let y_g denote the generated response and y_r the reference response. We combine four reward components, suppressing their shared arguments (y_g, y_r) for brevity:

$$R_{\text{tool}}(y_g, y_r) = R_{\text{fn}} + R_{\text{pn}} + R_{\text{pv}} + R_{\text{fmt}}. \tag{4}$$

Here, R_{fn}, R_{pn}, and R_{pv} evaluate function names, parameter names, and parameter values, respectively, and R_{fmt} evaluates output format. Let $S_{\text{fn}}, S_{\text{pn}}, S_{\text{pv}} \in [0, 1]$ denote their match-

Table 2: Tool-call results on RLLA-4K and API-Bank. API-Bank scores use the same all-exact metric for the correctness of final tool call. The rightmost Avg. is the arithmetic mean of RLLA-4K all-exact score and API-Bank Avg. score.

Model	RLLA-4K (In-dataset Eval)				API-Bank				Avg.
	Function	Param. Name	Param. Value	All Exact	V1	V2	V3	Avg.	
Qwen2.5-7B-Instruct	73.17	69.77	54.78	38.03	17.29	19.26	24.49	20.35	29.19
+ GRPO	**97.65**	**95.60**	76.91	63.38	79.45	44.44	**35.92**	53.27	58.33
+ CorrGRPO	95.07	94.95	**82.14**	**67.61**	**81.45**	**46.67**	35.10	**54.41**	**61.01**
Qwen3-4B-Thinking-2507	33.80	31.46	25.35	21.13	53.38	39.26	31.43	41.36	31.25
+ GRPO	**94.84**	92.08	72.07	57.75	**79.20**	52.59	52.24	61.34	59.55
+ CorrGRPO	93.78	**92.31**	**73.74**	**59.15**	78.45	**53.33**	**64.08**	**65.29**	**62.22**
Qwen3-8B	86.85	84.80	67.50	50.70	74.69	49.63	**46.53**	56.95	53.83
+ GRPO	92.72	91.43	72.54	61.97	78.20	51.85	40.82	56.95	59.46
+ CorrGRPO	**97.65**	**96.60**	**76.44**	**63.38**	**81.70**	**51.85**	44.08	**59.21**	**61.30**

ing scores. The reward components use the following scales: $R_{\text{fn}} = 0.5(2S_{\text{fn}} - 1)$, $R_{\text{pn}} = 2S_{\text{pn}} - 1$, $R_{\text{pv}} = 1.5(2S_{\text{pv}} - 1)$. These rewards provide partial credit for incomplete calls. Parameter matching requires a matching function, and value correctness requires the corresponding parameter name. Appendix J.2 provides the matching procedure and score definitions. The format reward is 1 when the response follows the required structure and 0 otherwise. We provide the response format template in Appendix J.4.

Experimental setting. We evaluate Qwen2.5-7B-Instruct (Yang et al., 2024), Qwen3-4B-Thinking-2507 (Qwen Team, 2025), and Qwen3-8B (Yang et al., 2025), comparing their base, GRPO, and CorrGRPO. We train on RLLA-4K from ToolRL (Qian et al., 2025), which contains user requests, tool descriptions, and reference responses specifying tool calls or direct answers. Our split contains 3,920 training examples and 80 test examples. To assess generalization without further training, we evaluate API-Bank (Li et al., 2023), a benchmark of tool-use dialogues with executable APIs. Table 2 reports normalized function-name, parameter-name, and parameter-value scores on RLLA-4K, with the all-exact score, the percentage of examples for which all tool-call fields are correct. API-Bank results cover v1, v2, and v3, with final-call correctness assessed using its execution-based checker.

Main Results. CorrGRPO improves complete-call correctness and generalization across all three evaluated backbones. Table 2 shows that CorrGRPO achieves the highest RLLA-4K and API-Bank average all-exact scores for every backbone. Compared with GRPO, RLLA-4K all-exact score increases from 63.38% to 67.61% for Qwen2.5-7B, from 57.75% to 59.15% for Qwen3-4B-Thinking, and from 61.97% to 63.38% for Qwen3-8B. These improvements accompany higher parameter-value scores across all three models, supporting more accurate argument generation. On API-Bank, the average score improves by 1.14, 3.94, and 2.26 percentage points, respectively. The largest gain occurs for Qwen3-4B-Thinking on API-Bank v3, where correctness increases from 52.24% to 64.08%. Together, these improvements raise the overall average by 2.68, 2.67, and 1.84 percentage points, demonstrating gains in both complete-call accuracy on the training-domain benchmark and transfer to API-Bank.

4.3 Agent Utility vs. Security

Task. We study tool-using agents that must complete legitimate user tasks while resisting indirect prompt injections embedded in tool outputs. An agent interacts with an environment through multiple tool calls, where malicious instructions may attempt to redirect its actions toward an attacker's objective. We evaluate both task completion and attack success to determine whether the agent remains useful under adversarial interference.

Reward design. For a valid attacked trajectory, let y_g include the agent's responses, tool calls, and observations, with prompt-injection content inserted into the tool observations. We assign equal weights to utility and security:

$$R_{\text{agent}}(y_g) = R_{\text{util}}(y_g) + R_{\text{sec}}(y_g). \tag{5}$$

Table 3: Agent utility and security results. Utility measures the agent's tool-calling success rate, while ASR measures the success rate of prompt injection attacks. Joint accuracy combines utility and security as $(1 - \text{attack_success}) \times \frac{\text{clean_utility+utility_under_attack}}{2}$. In InjecAgent, all reported scores represent ASR.

AgentDojo (In-dataset Eval) / Agent Security Bench

Model	Clean Utility ↑	Utility Under Attack ↑	ASR ↓	Joint Accuracy ↑
Qwen2.5-3B-Instruct	28.57 / 10.00	14.62 / 6.25	3.85 / **8.00**	18.97 / 6.13
+ GRPO	52.38 / 21.75	66.92 / 9.75	**0.00** / 9.00	60.13 / 14.88
+ CorrGRPO	**95.24** / **26.00**	**84.36** / **13.25**	1.54 / 9.25	**89.62** / **18.13**
Qwen2.5-7B-Instruct	38.10 / **64.75**	30.51 / 47.00	11.28 / 33.25	28.72 / 42.00
+ GRPO	**90.48** / 30.00	80.00 / 41.75	**0.26** / **26.50**	84.49 / 31.38
+ CorrGRPO	85.71 / 59.75	**83.08** / **53.25**	1.28 / 29.00	**86.03** / **47.88**
Qwen3-8B	71.43 / 60.00	32.31 / 51.50	19.49 / 31.25	40.38 / 49.63
+ GRPO	**85.71** / 60.00	**77.18** / 61.50	2.05 / **19.50**	**80.26** / 57.38
+ CorrGRPO	85.71 / **70.00**	75.13 / **61.75**	**1.54** / 22.50	79.62 / **58.63**

InjecAgent

Model	Direct Harm ↓	Data Stealing			Total ↓
		S1 ↓	S2 ↓	Total ↓	
Qwen2.5-3B-Instruct	18.38	50.29	76.39	34.59	27.12
+ GRPO	5.03	20.11	79.31	13.07	8.80
+ CorrGRPO	**4.67**	**17.26**	**52.63**	**6.33**	**5.52**
Qwen2.5-7B-Instruct	33.85	44.21	92.24	38.91	36.17
+ GRPO	16.49	15.06	75.00	10.12	13.53
+ CorrGRPO	**6.91**	**9.77**	**60.00**	**5.44**	**6.17**
Qwen3-8B	30.56	42.22	92.13	38.79	34.86
+ GRPO	13.45	16.36	88.75	14.37	13.94
+ CorrGRPO	**6.94**	**12.02**	**74.19**	**8.91**	**7.98**

The utility reward R_{util} is 1 if the agent successfully completes the user's task under attack and 0 otherwise. The security reward R_{sec} is 1 if the attacker's objective is not achieved and 0 otherwise. Both rewards are determined by the environment's task checkers. These objectives can compete: avoiding tool interactions may prevent an attack while also preventing completion of the user's task. Their combination rewards useful behavior and resistance to malicious instructions.

Experimental setting. We evaluate Qwen2.5-3B-Instruct, Qwen2.5-7B-Instruct (Yang et al., 2024), and Qwen3-8B (Yang et al., 2025), comparing their base, GRPO, and CorrGRPO variants. Training takes place in AgentDojo (Debenedetti et al., 2024). We construct a deterministic split grouped by user task, containing 1,584 training cases and 411 test cases; the latter comprise 21 clean cases and 390 attacked cases. To assess generalization without further training, we evaluate Agent Security Bench (ASB; Zhang et al., 2025) using 400 paired clean and attacked cases per model, and InjecAgent (Zhan et al., 2024). Table 3 reports clean utility, utility under attack, attack success rate (ASR), and Joint Accuracy for AgentDojo and ASB. For N paired cases, Joint Accuracy is computed as $\frac{1}{N}\sum_{i=1}^{N}(1 - A_i)\frac{U_{\text{clean},i}+U_{\text{attack},i}}{2}$, where $U_{\text{clean},i}$ and $U_{\text{attack},i}$ denote user-task success in the clean and attacked executions, and A_i denotes attack success in the attacked execution.

Main Results. CorrGRPO improves the balance between agent utility and security. Relative to GRPO, CorrGRPO increases ASB Joint Accuracy from 14.88% to 18.13% for Qwen2.5-3B, from 31.38% to 47.88% for Qwen2.5-7B, and from 57.38% to 58.63% for Qwen3-8B (Table 3). These gains accompany higher clean utility and utility under attack for all three models, supporting more effective task completion in adversarial environments. On InjecAgent, overall ASR decreases from 8.80% to 5.52%, from 13.53% to 6.17%, and from 13.94% to 7.98%, respectively, with reductions in every reported attack category. On AgentDojo, CorrGRPO also improves Joint Accuracy for Qwen2.5-3B and Qwen2.5-7B, with the largest gain at 3B, from 60.13% to 89.62%.

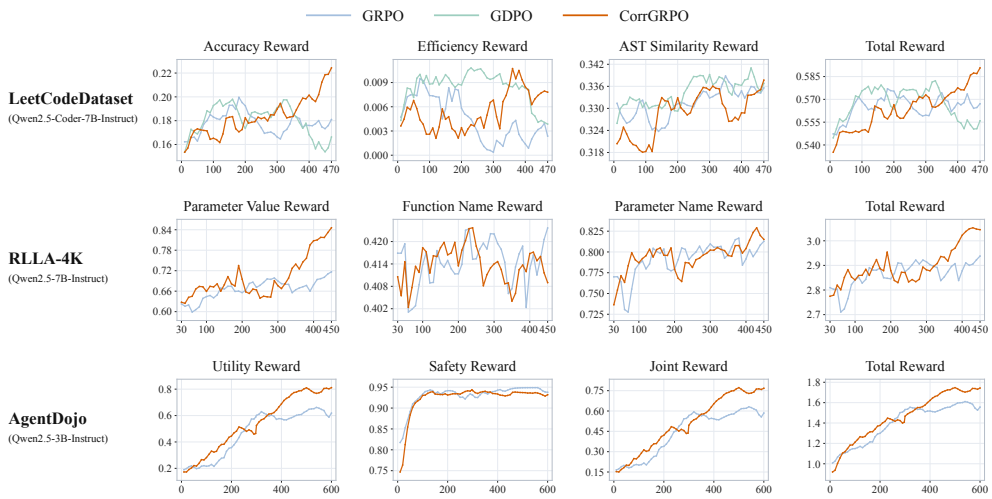


Figure 4: Validation reward dynamics of GRPO, GDPO, and CorrGRPO. Curves report validation mean@1 scores with exponential moving average smoothing (decay = 0.6).

5 Related Work

Multi-reward RL algorithms differ primarily in how they aggregate reward signals and balance their contributions during policy optimization. Scalarization-based approaches combine multiple rewards into a single training signal: MORLAIF trains separate preference models for individual objectives and aggregates their scores through scalarization functions before PPO updates (Williams, 2024). To adapt objective tradeoffs during training, Safe RLHF maximizes helpfulness subject to safety constraints using dynamically updated Lagrange multipliers (Dai et al., 2023), while dynamic reward weighting adjusts scalarization weights through hypervolume-guided adaptation or gradient-based optimization (Lu et al., 2025). More closely related to our work, recent methods modify how multiple rewards enter group-relative advantage estimation. MO-GRPO automatically reweights reward functions according to their variances to balance their contributions (Ichihara et al., 2025), whereas GDPO independently normalizes each reward within rollout groups, aggregates the resulting advantages, and applies batch-level normalization to stabilize their overall magnitude (Liu et al., 2026). RDPO further addresses reward dependence by combining magnitude-aware quantile normalization with Mahalanobis whitening before aggregation, reducing redundant variation among correlated reward dimensions (Bai et al., 2026). Our CorrGRPO instead retains the centered total reward and its prescribed relative weights, while replacing the covariance-based normalization denominator with an aggregate of Pearson correlations.

6 Conclusion

We introduced CorrGRPO, a correlation-normalized variant of GRPO for multi-reward reinforcement learning. Our analysis shows that normalizing the summed reward by its within-group standard deviation implicitly aggregates all pairwise reward covariances, coupling reward dependence with component scales. CorrGRPO replaces these covariance terms with Pearson correlation coefficients while preserving the centered total reward and its prescribed relative weights. This modification allows advantage normalization to respond to inter-reward correlations without being dominated by reward components with larger within-group standard deviations. Experiments on code generation, tool calling, and agent security, using models ranging from 0.5B to 8B parameters, demonstrate improved performance and an outward expansion of the empirical Pareto frontier between competing objectives. These results highlight the value of explicitly accounting for inter-reward correlations when designing advantage estimators for multi-reward learning.

AI Use Statement

We used generative AI tools to improve the clarity of expression, language quality, and grammatical correctness of the manuscript. Beyond the language models studied in our experiments, we did not use generative AI tools to develop the proposed method, formulate mathematical claims, write proofs, design experiments, implement methods, generate or process experimental datasets, or interpret results. The authors reviewed and revised all AI-assisted text to ensure accuracy and consistency with the underlying research. We take full responsibility for the final manuscript, including all claims, results, and AI-assisted content.

Ethics Statement

This work studies multi-reward optimization for language models through existing benchmarks for code generation, tool calling, and agent security. Improving these capabilities may benefit legitimate applications but may also facilitate harmful automation or unauthorized tool use. Our security experiments evaluate resistance to benchmark prompt-injection attacks; improvements on these benchmarks do not establish safety in real-world deployments or against unseen attacks. Moreover, correlation-based normalization does not correct biased or misspecified rewards, and the resulting behavior remains dependent on the selected objectives and their weights. Deployment therefore requires application-specific safety evaluation, appropriate tool-access restrictions, and human oversight.

Reproducibility Statement

Section 3 defines CorrGRPO, and Appendix B provides its core implementation, including numerical stabilization and the handling of zero-variance reward components. Sections 4.1–4.3 describe the model backbones, reward functions, experimental settings, and evaluation metrics. Appendix I reports training hyperparameters, Appendix J details reward computation, and Appendix M describes the datasets, evaluation subsets, and benchmark settings. Representative prompts and responses are provided in Appendix H. Appendix L presents the assumptions and derivations underlying the theoretical properties, while Appendix F details compatibility with other optimization methods. These materials support reproduction and inspection of the proposed method and its evaluation.

References

Jacob Austin et al. Program Synthesis with Large Language Models. *arXiv preprint arXiv:2108.07732*, 2021. URL https://arxiv.org/abs/2108.07732.

Yang Bai, Kaiyuan Liu, Ziyuan Zhuang, Jiahong Zhou, Rongxiang Weng, Xin Chen, Jingang Wang, and Xunliang Cai. Multi-objective and mixed-reward reinforcement learning via reward-decorrelated policy optimization. *arXiv preprint arXiv:2605.13641*, 2026. doi: 10.48550/arXiv.2605.13641. URL https://arxiv.org/abs/2605.13641.

Mark Chen et al. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374*, 2021. URL https://arxiv.org/abs/2107.03374.

Ganqu Cui, Yuchen Zhang, Jiacheng Chen, Lifan Yuan, Zhi Wang, Yuxin Zuo, Haozhan Li, Yuchen Fan, Huayu Chen, Weize Chen, Zhiyuan Liu, Hao Peng, Lei Bai, Wanli Ouyang, Yu Cheng, Bowen Zhou, and Ning Ding. The entropy mechanism of reinforcement learning for reasoning language models, 2025. URL https://arxiv.org/abs/2505.22617.

Josef Dai, Xuehai Pan, Ruiyang Sun, Jiaming Ji, Xinbo Xu, Mickel Liu, Yizhou Wang, and Yaodong Yang. Safe RLHF: Safe reinforcement learning from human feedback. *arXiv preprint arXiv:2310.12773*, 2023. doi: 10.48550/arXiv.2310.12773. URL https://arxiv.org/abs/2310.12773.

Edoardo Debenedetti, Jie Zhang, Mislav Balunović, Luca Beurer-Kellner, Marc Fischer, and Florian Tramèr. AgentDojo: A dynamic environment to evaluate prompt injection attacks and defenses for LLM agents. *arXiv preprint arXiv:2406.13352*, 2024. URL https://arxiv.org/abs/2406.13352.

DeepSeek-AI et al. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025. URL https://arxiv.org/abs/2501.12948.

Binyuan Hui et al. Qwen2.5-Coder Technical Report. *arXiv preprint arXiv:2409.12186*, 2024. URL https://arxiv.org/abs/2409.12186.

Yuki Ichihara, Yuu Jinnai, Tetsuro Morimura, Mitsuki Sakamoto, Ryota Mitsuhashi, and Eiji Uchibe. MO-GRPO: Mitigating reward hacking of group relative policy optimization on multi-objective problems. *arXiv preprint arXiv:2509.22047*, 2025. doi: 10.48550/arXiv.2509.22047. URL https://arxiv.org/abs/2509.22047.

Naman Jain et al. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. *arXiv preprint arXiv:2403.07974*, 2024. URL https://arxiv.org/abs/2403.07974.

Minghao Li et al. API-Bank: A Comprehensive Benchmark for Tool-Augmented LLMs. *arXiv preprint arXiv:2304.08244*, 2023. URL https://arxiv.org/abs/2304.08244.

Shih-Yang Liu et al. GDPO: Group reward-Decoupled Normalization Policy Optimization for Multi-reward RL Optimization. *arXiv preprint arXiv:2601.05242*, 2026. URL https://arxiv.org/abs/2601.05242.

Yining Lu, Zilong Wang, Shiyang Li, Xin Liu, Changlong Yu, Qingyu Yin, Zhan Shi, Zixuan Zhang, and Meng Jiang. Learning to optimize multi-objective alignment through dynamic reward weighting. *arXiv preprint arXiv:2509.11452*, 2025. doi: 10.48550/arXiv.2509.11452. URL https://arxiv.org/abs/2509.11452.

MiniMax. MiniMax-M1: Scaling test-time compute efficiently with lightning attention. *arXiv preprint arXiv:2506.13585*, 2025. URL https://arxiv.org/abs/2506.13585.

Long Ouyang et al. Training language models to follow instructions with human feedback. *arXiv preprint arXiv:2203.02155*, 2022. URL https://arxiv.org/abs/2203.02155.

Python Software Foundation. *Python Documentation: difflib—Helpers for Computing Deltas*. URL https://docs.python.org/3/library/difflib.html. Accessed 2026-09-14.

Cheng Qian et al. ToolRL: Reward is All Tool Learning Needs. *arXiv preprint arXiv:2504.13958*, 2025. URL https://arxiv.org/abs/2504.13958.

Qwen Team. Qwen3-4B-Thinking-2507. Model card, 2025. URL https://huggingface.co/Qwen/Qwen3-4B-Thinking-2507.

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

Marcus Williams. Multi-objective reinforcement learning from AI feedback. *arXiv preprint arXiv:2406.07295*, 2024. doi: 10.48550/arXiv.2406.07295. URL https://arxiv.org/abs/2406.07295.

Yunhui Xia et al. LeetCodeDataset: A Temporal Dataset for Robust Evaluation and Efficient Training of Code LLMs. *arXiv preprint arXiv:2504.14655*, 2025. URL https://arxiv.org/abs/2504.14655.

An Yang et al. Qwen2.5 Technical Report. *arXiv preprint arXiv:2412.15115*, 2024. URL https://arxiv.org/abs/2412.15115.

An Yang et al. Qwen3 Technical Report. *arXiv preprint arXiv:2505.09388*, 2025. URL https://arxiv.org/abs/2505.09388.

Qiying Yu, Zheng Zhang, Ruofei Zhu, et al. DAPO: An open-source LLM reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025. URL https://arxiv.org/abs/2503.14476.

Qiusi Zhan, Zhixiang Liang, Zifan Ying, and Daniel Kang. InjecAgent: Benchmarking Indirect Prompt Injections in Tool-Integrated Large Language Model Agents. *arXiv preprint arXiv:2403.02691*, 2024. URL https://arxiv.org/abs/2403.02691.

Hanrong Zhang et al. Agent Security Bench (ASB): Formalizing and Benchmarking Attacks and Defenses in LLM-based Agents. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2410.02644.

A Further Discussion

Preserving groupwise gradient directions. Let $L_{\mathrm{GRPO},q}$ and $L_{\mathrm{CorrGRPO},q}$ denote the clipped reward surrogates averaged over tokens and trajectories in a fixed rollout group for prompt q, excluding the KL term. Let $\mathbf{C} = [\hat{\rho}_{lm}]$ be the group's sample correlation matrix, $\mathbf{s} = (\hat{\sigma}_1, \ldots, \hat{\sigma}_r)^\top$ its vector of reward standard deviations, and $\mathbf{1}$ the all-ones vector. As derived in Appendix L.3, Equation 44, the shared numerator gives

$$A^i_{\mathrm{CorrGRPO}} = c_q A^i_{\mathrm{GRPO}}, \qquad c_q = \frac{\sqrt{\mathbf{s}^\top \mathbf{C}\mathbf{s}} + \varepsilon}{\sqrt{\mathbf{1}^\top \mathbf{C}\mathbf{1}} + \varepsilon} > 0. \tag{6}$$

The coefficient c_q is computed from the same group's reward statistics, shared across its trajectories, and held fixed during surrogate optimization. Because the clipped surrogate is positively homogeneous in the advantage, this scaling preserves the active clipping branches and yields

$$\nabla_\theta L_{\mathrm{CorrGRPO},q} = c_q \nabla_\theta L_{\mathrm{GRPO},q}, \qquad c_q > 0. \tag{7}$$

The derivation is given in Appendix L.3, Equations 48–50. Thus, at the same policy parameters, CorrGRPO preserves the direction of each group's nonzero reward-driven gradient while adapting its magnitude to reward dependence. The clipping thresholds remain unchanged. Since c_q can differ across groups, their relative contributions to the batch gradient can change; the KL term retains its original coefficient.

A quadratic-norm view of reward fluctuations. Using the same correlation matrix $\mathbf{C}$, define $\|\mathbf{x}\|_{\mathbf{C}} = \sqrt{\mathbf{x}^\top \mathbf{C}\mathbf{x}}$, which is a norm when $\mathbf{C}$ is positive definite and a seminorm when it is singular. The two normalization statistics use the same correlation metric with different input vectors:

$$\begin{aligned} \text{GRPO:} \quad & \widehat{\mathrm{Var}}(R) = \mathbf{s}^\top \mathbf{C}\mathbf{s} = \|\mathbf{s}\|^2_{\mathbf{C}}, \\ \text{CorrGRPO:} \quad & \sum_{l,m} \hat{\rho}_{lm} = \mathbf{1}^\top \mathbf{C}\mathbf{1} = \|\mathbf{1}\|^2_{\mathbf{C}}. \end{aligned} \tag{8}$$

Their advantage denominators are consequently $\|\mathbf{s}\|_{\mathbf{C}} + \varepsilon$ and $\|\mathbf{1}\|_{\mathbf{C}} + \varepsilon$, respectively.

This admits a signal-processing interpretation. Treat the standardized, centered reward components as correlated input channels to a linear combiner. For a channel-gain vector $\mathbf{x}$, the output's sample variance is $\mathbf{x}^\top \mathbf{C}\mathbf{x}$. Diagonal terms measure individual channel contributions, while off-diagonal terms capture reinforcement or cancellation between channels. GRPO uses $\mathbf{x} = \mathbf{s}$, restoring each source's own fluctuation amplitude before measuring the combined output. Its normalization therefore depends on both the correlation structure and the individual source amplitudes. CorrGRPO uses $\mathbf{x} = \mathbf{1}$, giving all standardized channels equal gain so that the combined fluctuation reflects their correlations without additional weighting by their original scales.

Compatibility with other policy optimization methods. CorrGRPO is compatible with other RL policy optimization methods, including DAPO (Yu et al., 2025), CISPO (MiniMax, 2025), and GDPO (Liu et al., 2026). For DAPO and CISPO, the original group-relative advantage can be directly replaced with A^i_{CorrGRPO}, while retaining their respective policy objectives and clipping operations. For GDPO, we retain its per-reward group normalization and aggregation, and replace only the final batch standard-deviation denominator with a correlation norm computed across the batch from the resulting per-reward advantage components. The corresponding objectives are derived in Appendices F.2.1, F.2.2, and F.2.3, respectively. We also evaluate this integration with CISPO on Qwen2.5-Coder-7B-Instruct, with results reported in Appendix F.1.

Slower entropy decay. Figure 12 shows that CorrGRPO preserves higher policy entropy over training, with slower entropy decay particularly evident on AgentDojo and LeetCode-Dataset. Although this pattern varies across settings, these results suggest that CorrGRPO can sustain policy entropy for longer during optimization. As shown by Cui et al. (2025), entropy collapse diminishes exploration and accompanies performance saturation, while entropy control enables continued exploration and improves downstream performance. From

this perspective, the higher entropy retained by CorrGRPO helps explain its stronger performance: maintaining policy diversity preserves opportunities to explore alternative reasoning paths and discover higher-reward responses, supporting continued improvement beyond initially successful behaviors.

B Core Implementation of CorrGRPO

We compare the advantage implementation for GRPO and CorrGRPO in Listing 1. For each prompt group, `group_scores` contains the $n \times r$ reward components with their configured weights already applied. The implementation uses sample covariance with divisor $n-1$, assigns zero rows and columns to zero-variance components, and clamps the correlation sum to zero to guard against negative round-off. The listing follows the source's numerical convention: CorrGRPO uses $\sqrt{\max(\sum_{l,m} \hat{\rho}_{lm}, 0) + \epsilon}$ with $\epsilon = 10^{-6}$, whereas the paper's formula and the GRPO implementation place ϵ outside the square root.

Listing 1: GRPO to CorrGRPO: abbreviated groupwise diff. Red lines are removed and green lines added; reward loading and group indexing are omitted.

```
--- GRPO
+++ CorrGRPO
 # group_scores: [n, r], weighted rewards; n > 1
 # Executed within torch.no_grad().
 centered_scores = group_scores - group_scores.mean(0, keepdim=True)
 centered_total_reward = centered_scores.sum(dim=-1)
-denominator = group_scores.sum(dim=-1).std() + epsilon
+covariance = centered_scores.T @ centered_scores / (group_scores.size(0) - 1)
+variances = covariance.diagonal()
+inverse_std = torch.where(
+    variances > 0,
+    torch.rsqrt(variances),
+    torch.zeros_like(variances),
+)
+covariance_coefficients = (
+    covariance * inverse_std[:, None] * inverse_std[None, :]
+)
+coefficient_sum = covariance_coefficients.sum().clamp_min(0)
+denominator = torch.sqrt(coefficient_sum + epsilon)
 group_advantages = centered_total_reward / denominator
 scalar_advantages.index_copy_(0, positions_tensor, group_advantages)

 advantages = scalar_advantages.unsqueeze(-1) * response_mask
 return advantages, advantages
```

C Correlation Analysis for Two Dependent Rewards

This section examines how two dependent rewards jointly shape CorrGRPO's advantage normalization as training progresses. We use runtime success and functional correctness in coding RL as an example, deriving their correlation and examining training dynamics on LeetCodeDataset.

Correlation analysis. Let $R_{\text{run}} \in \{0,1\}$ indicate whether a generated program executes without runtime errors, and $R_{\text{pass}} \in \{0,1\}$ indicate whether it passes all test cases. Since passing all tests requires successful execution, $R^i_{\text{run}} R^i_{\text{pass}} = R^i_{\text{pass}}$. Denoting their within-group means by $\bar{R}_{\text{run}}$ and $\bar{R}_{\text{pass}}$, their Pearson correlation, when both variances are nonzero, satisfies

$$\widehat{\rho}_{\text{run,pass}} = \frac{\widehat{\text{Cov}}(R_{\text{run}}, R_{\text{pass}})}{\sqrt{\widehat{\text{Var}}(R_{\text{run}})\widehat{\text{Var}}(R_{\text{pass}})}} = \frac{\bar{R}_{\text{pass}} - \bar{R}_{\text{run}}\bar{R}_{\text{pass}}}{\sqrt{\bar{R}_{\text{run}}(1-\bar{R}_{\text{run}})\bar{R}_{\text{pass}}(1-\bar{R}_{\text{pass}})}} = \sqrt{\frac{\bar{R}_{\text{pass}}(1-\bar{R}_{\text{run}})}{\bar{R}_{\text{run}}(1-\bar{R}_{\text{pass}})}}. \tag{9}$$

Once runtime success stabilizes at a fixed level below one, increasing the mean pass reward $\bar{R}_{\text{pass}}$ increases the correlation $\widehat{\rho}_{\text{run,pass}}$. With equal reward weights, CorrGRPO computes

$$A^i_{\text{CorrGRPO}} = \frac{(R^i_{\text{run}} - \bar{R}_{\text{run}}) + (R^i_{\text{pass}} - \bar{R}_{\text{pass}})}{\sqrt{2 + 2\widehat{\rho}_{\text{run,pass}}} + \varepsilon}. \tag{10}$$

As the two rewards become increasingly correlated, CorrGRPO enlarges the denominator and attenuates their combined signal for a fixed centered total reward, moderating the effect of redundant reward information on the policy update. In the limiting case $R_{\text{run}} = R_{\text{pass}}$, ignoring ε gives $A^i_{\text{CorrGRPO}} = R^i_{\text{run}} - \bar{R}_{\text{run}}$: duplicating the same reward does not double the advantage.

Experimental analysis. We train on LeetCodeDataset using only runtime-success and all-test passing rewards with equal weights, $R_{\text{code}} = R_{\text{run}} + R_{\text{pass}}$. As shown in Figure 5, runtime success improves rapidly and approximately converges first, remaining around 0.92–0.96 after about 100 steps. During this plateau, the pass rate continues to rise from approximately 0.36 to 0.54, while the reported reward correlation also increases overall. This trend qualitatively agrees with the analysis: as more executable programs pass all tests, the two reward signals increasingly overlap. Under CorrGRPO's normalization, stronger correlation increases the denominator and reduces the shared advantage scale, moderating the reinforcement from these overlapping signals while preserving their equal weights.

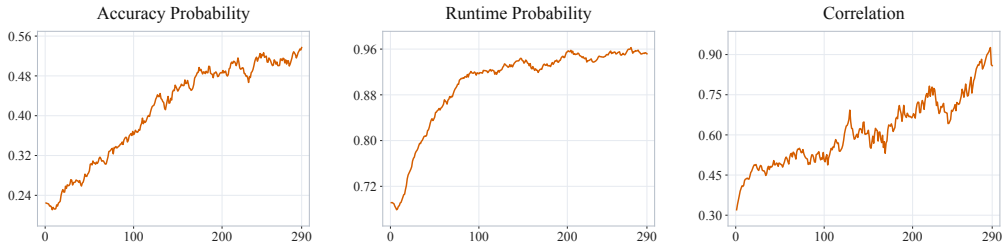


Figure 5: Training dynamics on LeetCodeDataset using equally weighted runtime-success and passing rewards.

D Additional Training Dynamics

We provide complete training curves for the experimental settings in the main paper. Validation and training rewards are shown for LeetCodeDataset in Figures 6 and 7, RLLA-4K in Figures 8 and 9, and AgentDojo in Figures 10 and 11. Figure 12 presents the mean response length and policy entropy across all three datasets. All horizontal axes indicate training steps, and all curves use exponential moving average smoothing with a decay factor of 0.6.

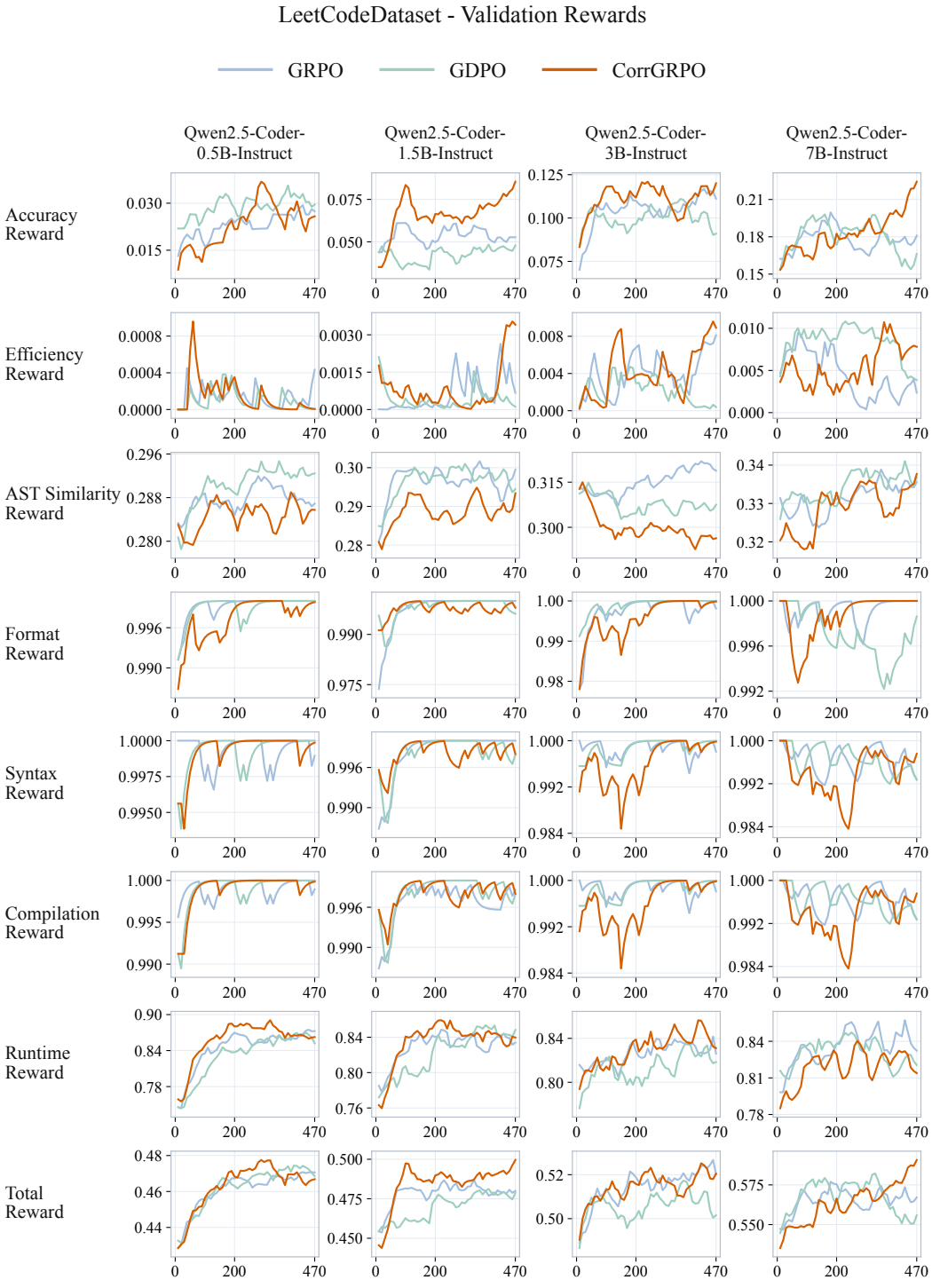


Figure 6: LeetCodeDataset: validation rewards. Columns correspond to model backbones. Rows report the task-specific validation mean@1 reward components and the total reward. The horizontal axis denotes training steps. Curves use exponential moving average smoothing (decay 0.6) and end at the last available observation.

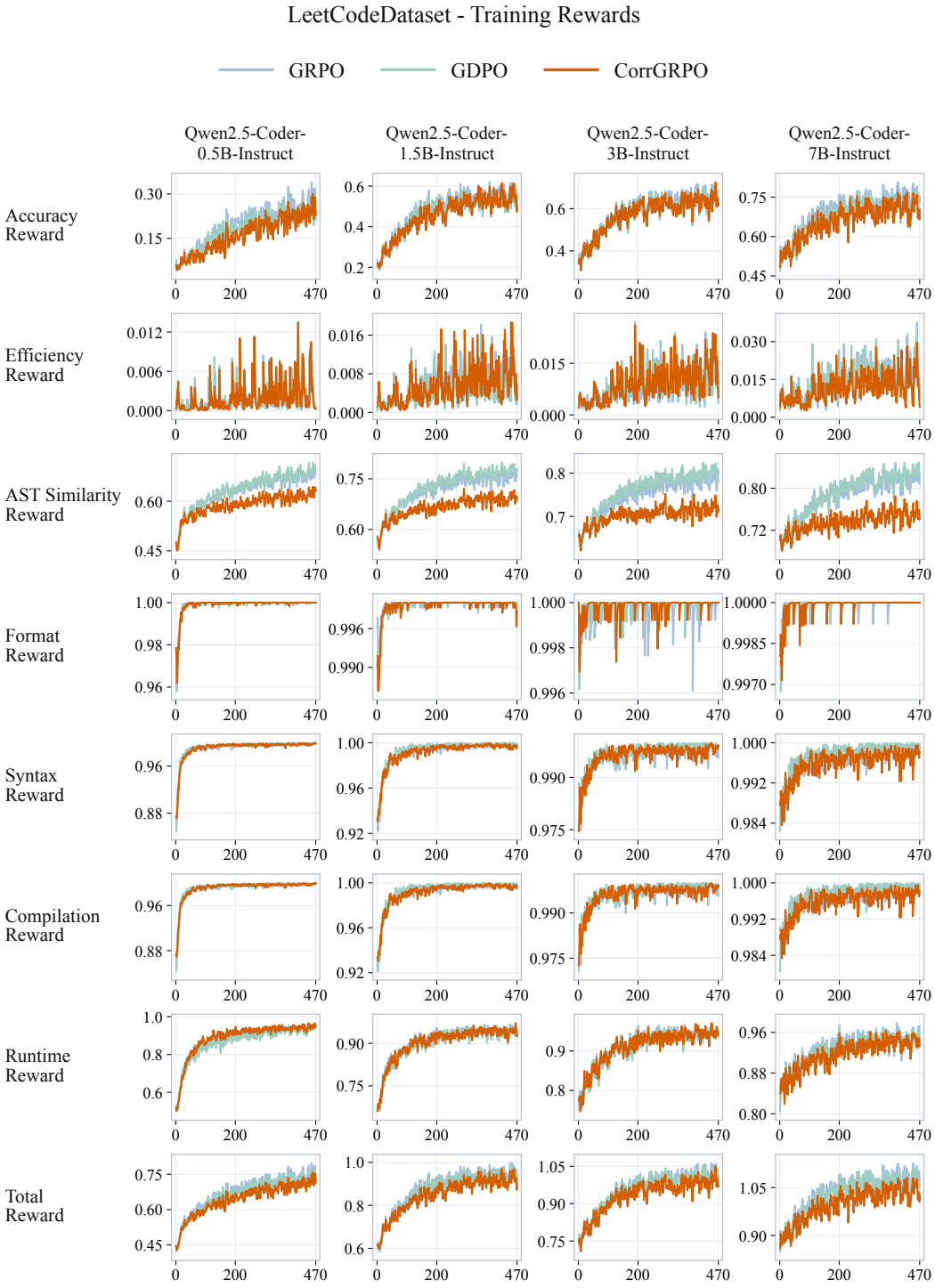


Figure 7: LeetCodeDataset: training rewards. Columns correspond to model backbones. Rows report the mean training reward components and the mean total reward. The horizontal axis denotes training steps. Curves use exponential moving average smoothing (decay 0.6) and end at the last available observation.



Figure 8: RLLA-4K: validation rewards. Columns correspond to model backbones. Rows report the task-specific validation mean@1 reward components and the total reward. The horizontal axis denotes training steps. Curves use exponential moving average smoothing (decay 0.6) and end at the last available observation.

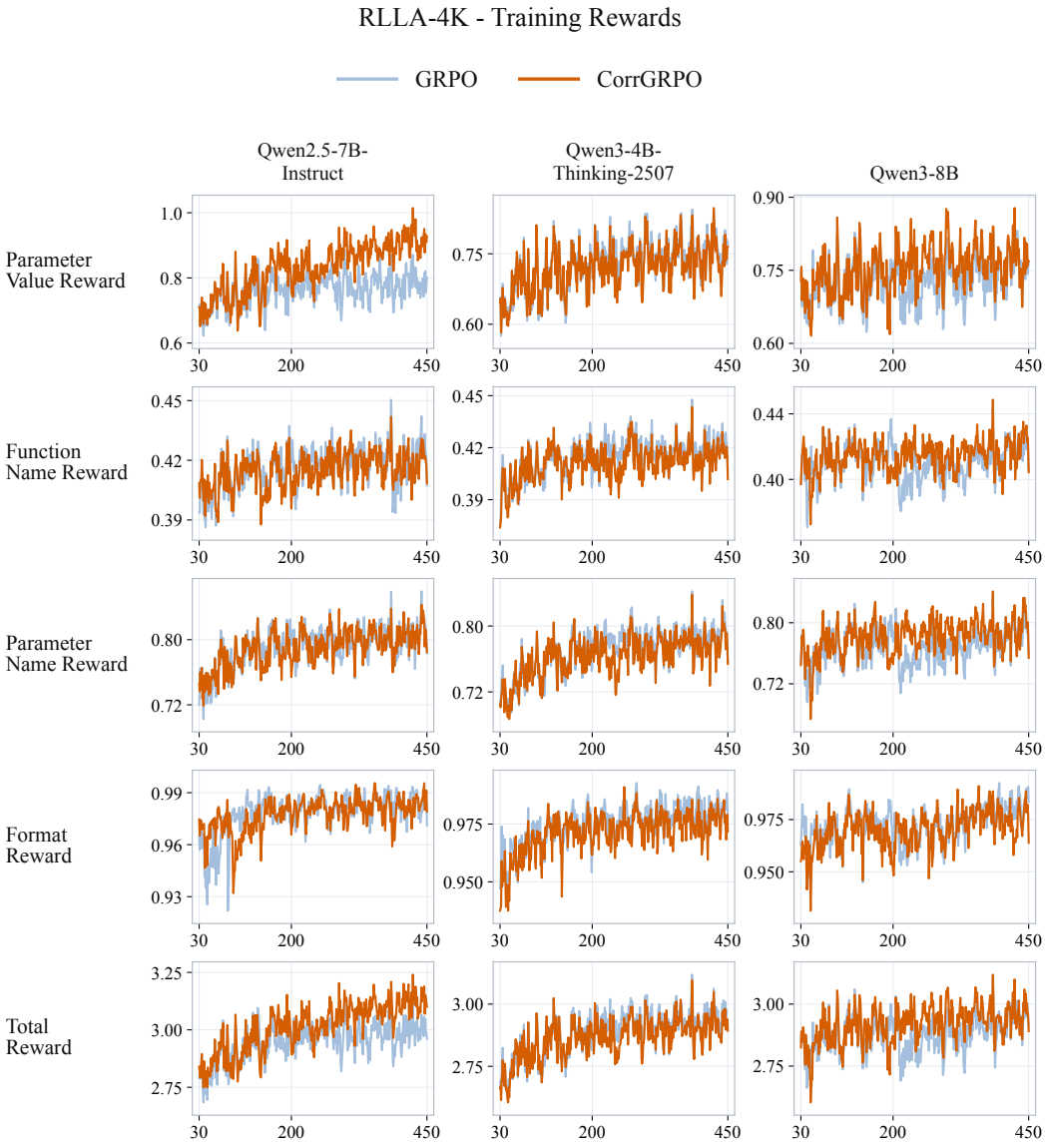


Figure 9: RLLA-4K: training rewards. Columns correspond to model backbones. Rows report the mean training reward components and the mean total reward. The horizontal axis denotes training steps. Curves use exponential moving average smoothing (decay 0.6) and end at the last available observation.

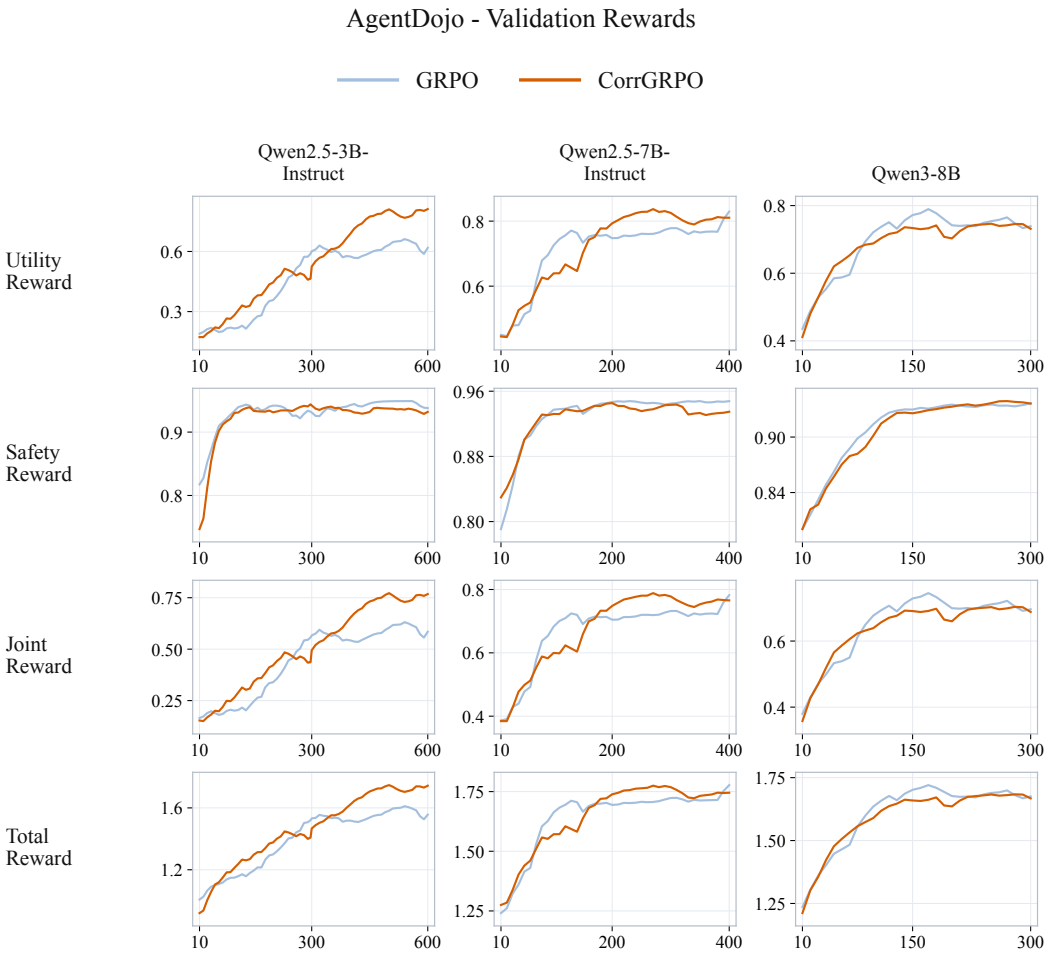


Figure 10: AgentDojo: validation rewards. Columns correspond to model backbones. Rows report the task-specific validation mean@1 reward components and the total reward. Joint Reward denotes the logged trajectory-level metric. The horizontal axis denotes training steps. AgentDojo panels are restricted to the overlapping recorded step ranges of GRPO and CorrGRPO. Curves use exponential moving average smoothing (decay 0.6) and end at the last available observation.

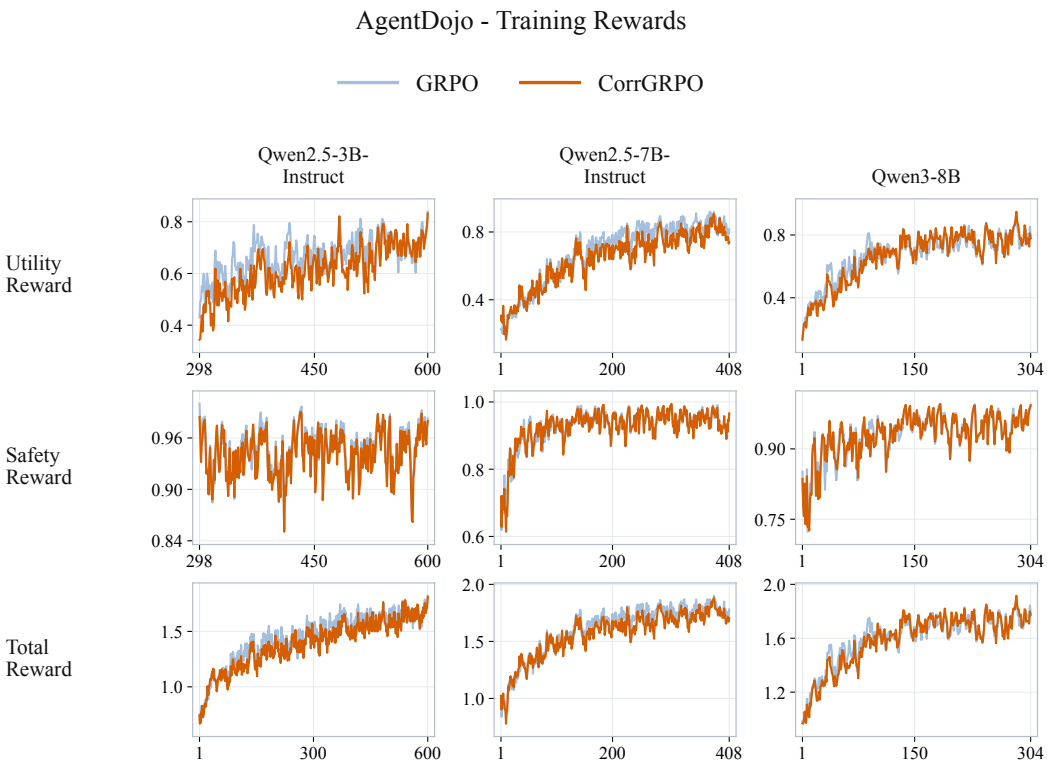


Figure 11: AgentDojo: training rewards. Columns correspond to model backbones. Rows report the mean training reward components and the mean total reward. The horizontal axis denotes training steps. AgentDojo panels are restricted to the overlapping recorded step ranges of GRPO and CorrGRPO. Curves use exponential moving average smoothing (decay 0.6) and end at the last available observation.

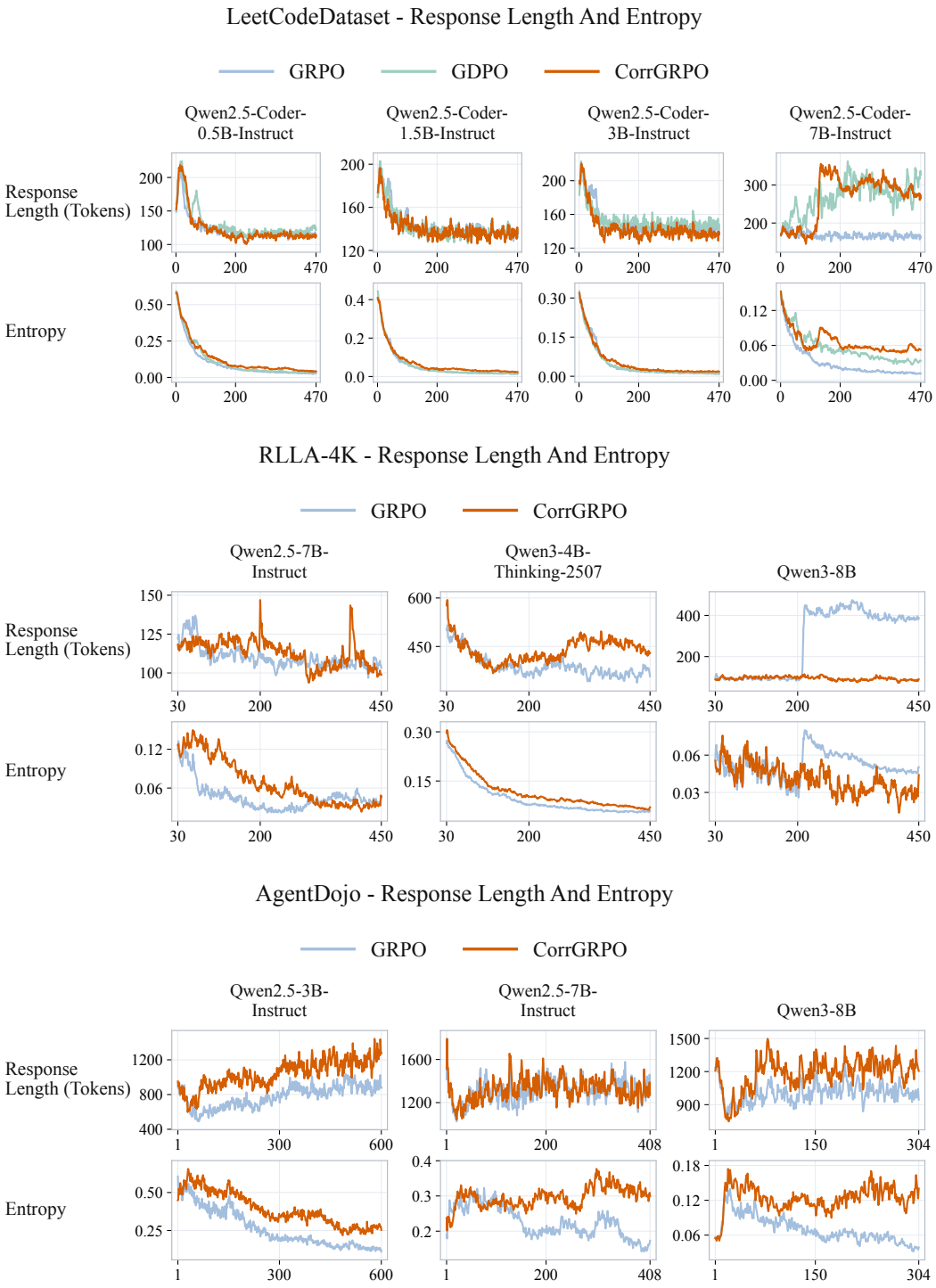


Figure 12: Response length and entropy on LeetCodeDataset (top), RLLA-4K (middle), and AgentDojo (bottom). Within each dataset, columns correspond to model backbones; the two rows show mean response length in tokens and policy entropy, respectively. Horizontal axes denote training steps. AgentDojo panels are restricted to the overlapping recorded step ranges of GRPO and CorrGRPO. Curves use exponential moving average smoothing (decay 0.6) and end at the last available observation within the displayed range.

E Reference Scores from the Original Benchmark Papers

Tables 4–7 report scores from the original benchmark papers alongside our base model, GRPO, and CorrGRPO. All scores are percentages. Evaluation protocols differ across papers.

E.1 Code Generation

The original SFT scores in Table 4 are taken from Xia et al. (2025, Table 4). The reference SFT runs use Qwen2.5-Coder-7B; our RL runs start from Qwen2.5-Coder-7B-Instruct and train on the 2,641-problem LeetCodeDataset split. We evaluate Pass@1 on LeetCodeDataset, HumanEval, MBPP, and LiveCodeBench v6 using the evaluation splits listed below.

Table 4: Code-generation Pass@1. Evaluation splits are indicated in the table.

Training data / method	Training examples	HumanEval ↑	MBPP ↑	LiveCode Bench ↑	LeetCode Dataset ↑
Original SFT results: Qwen2.5-Coder-7B					
				24-08–25-02	24-07–25-03
Magicoder Evol-Instruct-110K	111.1K	77.4	74.1	15.1	13.7
Magicoder OSS-Instruct-75K	75.1K	73.8	76.5	15.1	12.9
Open-R1 CodeForces-CoT	9.5K	79.9	74.1	15.8	13.3
OpenThoughts 114k	19.9K	77.4	75.7	16.9	16.4
LeetCodeDataset (human)	2.6K	55.5	53.4	14.0	10.9
LeetCodeDataset (model)	2.6K	79.9	77.5	15.4	12.5
Ours: Qwen2.5-Coder-7B-Instruct; LeetCodeDataset training					
				v6, 175 tasks	228-problem split
Base model	—	78.66	76.40	20.57	14.91
+GRPO	2,641	73.78	78.40	21.14	15.79
+CorrGRPO	2,641	78.66	78.60	24.57	24.12

E.2 AgentDojo

The original scores in Table 5 are taken from Debenedetti et al. (2024, Table 3). We train Qwen2.5-3B-Instruct on 1,584 cases with a split grouped by user task, and evaluate on 21 clean and 390 attacked cases. We inject `important_instructions` or `tool_knowledge` via tool responses.

Table 5: AgentDojo with original 95% confidence intervals; protocols differ.

Model	Clean utility ↑	Utility under attack ↑	Targeted ASR ↓
Claude 3 Opus	66.61 ± 3.69	52.46 ± 3.90	11.29 ± 2.47
Claude 3 Sonnet	53.10 ± 3.90	33.23 ± 3.68	26.71 ± 3.46
Claude 3.5 Sonnet	78.22 ± 3.23	51.19 ± 3.91	33.86 ± 3.70
Command-R+	25.44 ± 3.40	25.12 ± 3.39	0.95 ± 0.76
Gemini 1.5 Flash	36.09 ± 3.75	34.18 ± 3.71	12.24 ± 2.56
Gemini 1.5 Pro	45.63 ± 3.89	28.93 ± 3.54	25.60 ± 3.41
GPT-3.5 Turbo	33.86 ± 3.70	34.66 ± 3.72	8.43 ± 2.17
GPT-4 Turbo	63.43 ± 3.76	54.05 ± 3.89	28.62 ± 3.53
GPT-4o	69.00 ± 3.61	50.08 ± 3.91	47.69 ± 3.90
Llama 3 70B	34.50 ± 3.71	18.28 ± 3.02	20.03 ± 3.13
Ours: Qwen2.5-3B-Instruct, held-out evaluation split			
Base model	28.57	14.62	3.85
+GRPO	52.38	66.92	0.00
+CorrGRPO	95.24	84.36	1.54

E.3 Agent Security Bench

The original OPI ASR and Clean Utility values in Table 6 are taken from Tables 5 and 6 of Zhang et al. (2025), respectively. We evaluate the same Qwen2.5-3B-Instruct checkpoints on 400 paired clean and attacked cases without further training. Attacks append `context_ignoring` payloads to intermediate tool responses.

Table 6: Clean Utility and OPI ASR on Agent Security Bench.

Model	Clean utility ↑	ASR (OPI) ↓
Claude-3.5 Sonnet	100.00	59.70
LLaMA3-70B	66.50	43.70
GPT-4o	79.00	62.45
Gemma2-27B	31.50	14.20
LLaMA3.1-70B	21.25	12.10
Qwen2-7B	9.75	9.00
Gemma2-9B	10.75	14.20
GPT-3.5 Turbo	8.00	55.10
Qwen2-72B	4.00	21.35
LLaMA3.1-8B	0.75	6.40
Mixtral-8x7B	0.00	4.80
Ours: Qwen2.5-3B-Instruct		
Base model	10.00	8.00
+GRPO	21.75	9.00
+CorrGRPO	26.00	9.25

E.4 InjecAgent

The original base-setting scores in Table 7 are taken from Zhan et al. (2024, Table 3). We use the base setting and standard InjecAgent prompt without further training. The agent continues from an injected tool response; ASR-valid excludes invalid outputs.

Table 7: InjecAgent base-setting ASR-valid. S1/S2 denote data extraction/transmission; S2 is conditional on reaching that stage.

Model	Base setting: ASR-valid ↓				
	Direct harm	Data stealing			Overall
		S1	S2	Total	total
Original prompted agents (ReAct)					
Qwen-1.8B	36.1	35.1	82.6	17.6	29.7
Qwen-72B	8.7	37.9	98.4	37.1	23.2
Mistral-7B	13.4	25.0	87.8	20.1	16.7
OpenOrca-Mistral	3.9	5.3	53.8	2.9	3.4
OpenHermes-2.5-Mistral	23.4	29.2	99.2	28.4	25.9
Mixtral-8x7B	23.1	34.1	99.1	32.9	27.8
Nous-Mixtral-DPO	37.2	51.6	98.4	50.5	43.6
Nous-Mixtral-SFT	51.8	48.2	98.9	47.5	49.8
Platypus2-70B	34.3	51.8	74.3	35.4	34.9
Llama2-70B	91.9	97.1	83.7	80.4	86.9
Claude-2	7.5	26.5	58.1	14.8	11.4
GPT-3.5	18.8	37.6	77.4	28.8	23.7
GPT-4	14.7	32.7	97.7	31.9	23.6
Ours: Qwen2.5-3B-Instruct; RL training on AgentDojo					
Base model	18.38	50.29	76.39	34.59	27.12
+GRPO	5.03	20.11	79.31	13.07	8.80
+CorrGRPO	4.67	17.26	52.63	6.33	5.52

F Compatibility with CISPO, GDPO, and DAPO

F.1 Compatibility Experiments

We evaluate CorrGRPO in combination with CISPO, GDPO, and DAPO to examine its applicability across different policy optimization methods. Table 8 compares each method with its CorrGRPO-integrated variant. We report Efficiency, Executable, and Pass@1 on LeetCodeDataset, together with Pass@1 on HumanEval, MBPP, and LiveCodeBench v6. The average summarizes Pass@1 across the four benchmarks. These results assess both performance on the training-domain benchmark and generalization to external benchmarks. For DAPO, group filtering is based on the accuracy reward.

Table 8: Results of compatibility RL experiments.

Model	LeetCodeDataset (In-dataset Eval)			HumanEval	MBPP	LCB v6	Avg.
	Efficiency	Executable	Pass@1	Pass@1	Pass@1	Pass@1	Pass@1
CISPO	34.51	83.77	21.93	76.83	79.00	23.43	50.30
CISPO + CorrGRPO	38.44	82.46	25.00	81.71	77.80	21.71	51.56
GDPO	41.28	84.21	16.23	76.83	79.60	22.86	48.88
GDPO + CorrGRPO	44.64	86.40	17.11	76.83	77.20	21.71	48.21
DAPO	40.48	81.14	20.18	81.71	77.40	23.43	50.68
DAPO + CorrGRPO	47.07	82.89	21.93	80.49	78.40	22.86	50.92

F.2 Compatibility Analysis

We derive the integrations on fixed sampled data. All advantages and normalization statistics are held fixed during policy differentiation. The objectives below are reward surrogates; any separately configured KL or other auxiliary term retains its original definition. For a group associated with prompt q, write $T_q = \sum_{i=1}^{n} T_i$ and use the policy ratios $u_{i,t}(\theta)$ defined in the main text. Numerical stabilizers are made explicit using the same convention as CorrGRPO.

F.2.1 DAPO

DAPO (Yu et al., 2025) uses an asymmetric clipped surrogate with token-level averaging. In our notation, its group contribution is

$$\begin{aligned} L_{\mathrm{DAPO},q}(\theta; A) &= \frac{1}{T_q} \sum_{i=1}^{n} \sum_{t=1}^{T_i} \min\left\{ u_{i,t}(\theta) A^i, \kappa_D(u_{i,t}(\theta)) A^i \right\}, \\ \kappa_D(u) &= \mathrm{clip}(u, 1-\eta_{\mathrm{low}}, 1+\eta_{\mathrm{high}}). \end{aligned} \tag{11}$$

The baseline uses $A^i = A^i_{\mathrm{GRPO}}$. Substituting the CorrGRPO advantage gives the combined objective

$$L_{\mathrm{DAPO+CorrGRPO},q}(\theta) = \frac{1}{T_q} \sum_{i=1}^{n} \sum_{t=1}^{T_i} \min\left\{ u_{i,t}(\theta) A^i_{\mathrm{CorrGRPO}}, \kappa_D(u_{i,t}(\theta)) A^i_{\mathrm{CorrGRPO}} \right\}. \tag{12}$$

For c_q in Equation 44, positive homogeneity of the minimum gives

$$\begin{aligned} L_{\mathrm{DAPO+CorrGRPO},q}(\theta) &= c_q L_{\mathrm{DAPO},q}(\theta; A_{\mathrm{GRPO}}), \\ \nabla_\theta L_{\mathrm{DAPO+CorrGRPO},q} &= c_q \nabla_\theta L_{\mathrm{DAPO},q}(\theta; A_{\mathrm{GRPO}}). \end{aligned} \tag{13}$$

The asymmetric thresholds and token averaging are preserved. DAPO's sampling filter and reward shaping can be applied before this substitution using the same configured rules. The equality compares the same retained group at the same policy parameters; subsequent sampled groups can differ as training proceeds. The expected training objective averages these group contributions over the sampling procedure.

F.2.2 CISPO

CISPO (MiniMax, 2025) clips importance-sampling weights and stops gradients through those weights. Define

$$\tilde{u}_{i,t}(\theta) = \mathrm{clip}\big(u_{i,t}(\theta), 1-\eta^{\mathrm{IS}}_{\mathrm{low}}, 1+\eta^{\mathrm{IS}}_{\mathrm{high}}\big)\,, \tag{14}$$

where sg below denotes stop-gradient. The CISPO surrogate is

$$L_{\mathrm{CISPO},q}(\theta; A) = \frac{1}{T_q}\sum_{i=1}^{n}\sum_{t=1}^{T_i} \mathrm{sg}[\tilde{u}_{i,t}(\theta)]\, A^i \log \pi_\theta(a_{i,t} \mid h_{i,t}). \tag{15}$$

Its original group-relative advantage can be replaced directly:

$$L_{\mathrm{CISPO+CorrGRPO},q}(\theta) = \frac{1}{T_q}\sum_{i=1}^{n}\sum_{t=1}^{T_i} \mathrm{sg}[\tilde{u}_{i,t}(\theta)]\, A^i_{\mathrm{CorrGRPO}} \log \pi_\theta(a_{i,t} \mid h_{i,t}). \tag{16}$$

Differentiating with the prescribed stop-gradient operation yields

$$\begin{aligned}\nabla_\theta L_{\mathrm{CISPO+CorrGRPO},q} &= \frac{1}{T_q}\sum_{i=1}^{n}\sum_{t=1}^{T_i} \mathrm{sg}[\tilde{u}_{i,t}(\theta)] A^i_{\mathrm{CorrGRPO}} \nabla_\theta \log \pi_\theta(a_{i,t} \mid h_{i,t}) \\ &= c_q \nabla_\theta L_{\mathrm{CISPO},q}(\theta; A_{\mathrm{GRPO}}).\end{aligned} \tag{17}$$

Thus, the replacement preserves CISPO's importance-weight clipping and stop-gradient computation, while rescaling each group's reward-driven contribution. It introduces no PPO-style minimum or additional token-dropping rule. Table 8 reports the supplied CISPO checkpoint comparison; the mathematical construction does not assume that the two checkpoints were trained for equal numbers of steps.

F.2.3 GDPO

GDPO (Liu et al., 2026) first normalizes each reward within its prompt group, aggregates those components, and then normalizes the aggregate over the batch. Let $\mathcal{B}$ contain N sampled trajectories, with $q(b)$ denoting the prompt group of trajectory b. Write its component advantages as

$$U^b_l = w_l \frac{R^b_l - \bar{R}_{l,q(b)}}{\hat{\sigma}_{l,q(b)} + \varepsilon_g}, \qquad H^b = \sum_{l=1}^{r} U^b_l, \qquad \bar{H}_{\mathcal{B}} = \frac{1}{N}\sum_{b\in\mathcal{B}} H^b. \tag{18}$$

Here w_l denotes any weight applied after group normalization, with $w_l = 1$ for unweighted aggregation; R_l is the input to GDPO's per-reward normalization. The first-stage stabilizer ε_g is retained if present in the baseline implementation. GDPO's final advantage is

$$A^b_{\mathrm{GDPO}} = \frac{H^b - \bar{H}_{\mathcal{B}}}{\sqrt{\widehat{\mathrm{Var}}_{\mathcal{B}}(H)} + \varepsilon_b}. \tag{19}$$

The batch standard deviation in this expression is computed from H, so its covariance decomposition concerns the components U_l already produced by GDPO. Define their batch covariance matrix and standard-deviation vector as

$$\begin{aligned}\mathbf{\Sigma}_{\mathcal{B}} = \Big[\widehat{\mathrm{Cov}}_{\mathcal{B}}(U_l, U_m)\Big]_{l,m}, \qquad \mathbf{s}_{\mathcal{B}} = \left(\sqrt{\widehat{\mathrm{Var}}_{\mathcal{B}}(U_l)}\right)_{l=1}^{r}, \\ [\mathbf{C}_{\mathcal{B}}]_{lm} = \frac{[\mathbf{\Sigma}_{\mathcal{B}}]_{lm}}{[\mathbf{s}_{\mathcal{B}}]_l [\mathbf{s}_{\mathcal{B}}]_m}.\end{aligned} \tag{20}$$

By the finite-sample identity in Appendix L.2, Equation 42,

$$\widehat{\mathrm{Var}}_{\mathcal{B}}(H) = \mathbf{1}^\top \mathbf{\Sigma}_{\mathcal{B}} \mathbf{1} = \mathbf{s}_{\mathcal{B}}^\top \mathbf{C}_{\mathcal{B}} \mathbf{s}_{\mathcal{B}}. \tag{21}$$

Applying our normalization at this final batch stage gives

$$A^b_{\text{GDPO+CorrGRPO}} = \frac{H^b - \bar{H}_{\mathcal{B}}}{\sqrt{\mathbf{1}^\top \mathbf{C}_{\mathcal{B}} \mathbf{1}} + \varepsilon_b}. \tag{22}$$

The group normalization, aggregation weights, and batch-centered numerator are preserved. Only the last denominator changes, from $\|\mathbf{s}_{\mathcal{B}}\|_{\mathbf{C}_{\mathcal{B}}} + \varepsilon_b$ to $\|\mathbf{1}\|_{\mathbf{C}_{\mathcal{B}}} + \varepsilon_b$. Correlations are computed across the batch between the per-reward advantages U_l, using the same sample scope and covariance convention as the original batch statistic. An implementation that uses token masks or weighted batch moments must use the same masks or weights for every covariance and variance in this step.

Using the clipped token surrogate ℓ defined in Equation 45, a sequence-averaged objective is

$$L_{\text{GDPO+CorrGRPO},\mathcal{B}}(\theta) = \frac{1}{N} \sum_{b \in \mathcal{B}} \frac{1}{T_b} \sum_{t=1}^{T_b} \ell\big(u_{b,t}(\theta), A^b_{\text{GDPO+CorrGRPO}}\big). \tag{23}$$

If the baseline uses another fixed token reduction or asymmetric clipping, that choice is retained. Since the new and old advantages differ by the same positive factor for the entire batch,

$$\begin{aligned} c_{\mathcal{B}} &= \frac{\sqrt{\mathbf{s}_{\mathcal{B}}^\top \mathbf{C}_{\mathcal{B}} \mathbf{s}_{\mathcal{B}}} + \varepsilon_b}{\sqrt{\mathbf{1}^\top \mathbf{C}_{\mathcal{B}} \mathbf{1}} + \varepsilon_b} > 0, \\ A^b_{\text{GDPO+CorrGRPO}} &= c_{\mathcal{B}} A^b_{\text{GDPO}}. \end{aligned} \tag{24}$$

the clipped reward gradients satisfy

$$\nabla_\theta L_{\text{GDPO+CorrGRPO},\mathcal{B}} = c_{\mathcal{B}} \nabla_\theta L_{\text{GDPO},\mathcal{B}}. \tag{25}$$

This substitution can reduce to a common scale correction when GDPO's component normalization already equalizes batch variances. Specifically, if $\mathbf{s}_{\mathcal{B}} = s_0 \mathbf{1}$, then $c_{\mathcal{B}} = (s_0 \sqrt{\mathbf{1}^\top \mathbf{C}_{\mathcal{B}} \mathbf{1}} + \varepsilon_b)/(\sqrt{\mathbf{1}^\top \mathbf{C}_{\mathcal{B}} \mathbf{1}} + \varepsilon_b)$, approximately s_0 when the stabilizer is negligible. The integration establishes compatibility, rather than a universal additional benefit over GDPO. As elsewhere, the correlation formulas apply to nonconstant components; batch-constant components contribute zero after batch centering and can be omitted from the correlation matrix.

G Reinforcement Learning Background

We consider reinforcement learning for a language model policy π_θ. Given a prompt $q \sim \mathcal{D}$, the policy generates a trajectory τ, which may include a response or a sequence of interactions with an environment. Each trajectory is evaluated by r reward components $R_1(q,\tau), \ldots, R_r(q,\tau)$, capturing different aspects of the desired behavior. The learning objective is to maximize the expected total reward:

$$\max_\theta J(\theta) = \mathbb{E}_{q \sim \mathcal{D},\, \tau \sim \pi_\theta(\cdot|q)} \left[R(q,\tau) \right], \qquad R(q,\tau) = \sum_{l=1}^{r} R_l(q,\tau). \tag{26}$$

Any fixed reward weights are absorbed into the corresponding components. These components may exhibit positive or negative statistical dependence across trajectories, reflecting outcomes that tend to improve together or involve tradeoffs.

GRPO estimates advantages using rewards from a group of trajectories, avoiding a separate value model. For each prompt q, it samples n trajectories $\{\tau_i\}_{i=1}^n$ from an old policy $\pi_{\theta_{\text{old}}}$. Let $R_l^i = R_l(q,\tau_i)$ and $R^i = \sum_l R_l^i$, with group means $\bar{R}_l = \frac{1}{n}\sum_i R_l^i$ and $\bar{R} = \frac{1}{n}\sum_i R^i$. Under outcome supervision, all generated tokens in trajectory i share the same advantage A^i_{GRPO}. GRPO optimizes the clipped surrogate objective

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}\left[\frac{1}{n} \sum_{i=1}^{n} \frac{1}{T_i} \sum_{t=1}^{T_i} \Big(\min\Big[u_{i,t}(\theta) A^i_{\text{GRPO}}, \text{clip}(u_{i,t}(\theta), 1-\eta, 1+\eta)\, A^i_{\text{GRPO}} \Big] - \beta \mathcal{K}_{i,t} \Big) \right], \tag{27}$$

Here, the expectation is over prompts and trajectory groups sampled as described above. The ratio $u_{i,t}(\theta) = \pi_\theta(a_{i,t} \mid h_{i,t})/\pi_{\theta_{\text{old}}}(a_{i,t} \mid h_{i,t})$ is the policy probability ratio, $h_{i,t}$ is the history preceding generated token $a_{i,t}$, T_i counts generated tokens, η is the clipping threshold, and $\mathcal{K}_{i,t}$ is the KL regularization term relative to a reference policy.

H Examples from the Training Data

We present one example from each training dataset. The coding and tool-calling examples pair the training prompt with its stored reference response. AgentDojo supplies an interactive task and environment rather than a fixed reference response; we therefore show a recorded agent interaction for a case in our training split. This historical rollout illustrates the data format and is not a CorrGRPO training rollout or a comparative result.

H.1 Coding: LeetCodeDataset

The example is `remove-vowels-from-a-string` from the training split. The prompt below is the actual user message supplied to the model, including the problem description and required answer format. The response is the stored dataset reference; the code extracted from it is also used as the reference solution in the RL record.

Prompt (user).

````
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.

### Question:
Given a string s, remove the vowels 'a', 'e', 'i', 'o', and 'u' from it, and
    return the new string.

Example 1:

Input: s = "leetcodeisacommunityforcoders"
Output: "ltcdscmmntyfrcdrs"

Example 2:

Input: s = "aeiou"
Output: ""


Constraints:

1 <= s.length <= 1000
s consists of only lowercase English letters.



### Format: You will use the following starter code to write the solution to the
    problem and enclose your code within delimiters.
```python
class Solution:
    def removeVowels(self, s: str) -> str:

```

### Answer: (use the provided format with backticks)
````

Response (dataset reference).

````
```python
class Solution:
    def removeVowels(self, s: str) -> str:
        vowels = set('aeiou')
        return ''.join([char for char in s if char not in vowels])
```
````

H.2 Tool Calling: RLLA-4K

This training example asks for the least common multiple of two integers. The system message specifies the available function, its argument schema, and the response format. We reproduce both prompt messages and the stored reference response, including its original `<think>` and `<tool_call>` fields.

Prompt (system).

````
You are a helpful multi-turn dialogue assistant capable of leveraging tool calls
    to solve user tasks and provide structured chat responses.

**Available Tools**
In your response, you can use the following tools:
1. Name: least_common_multiple
Description: Computes the least common multiple (LCM) of two positive integers.
Parameters: {"a": {"description": "The first positive integer.", "type": "int", "
    default": ""}, "b": {"description": "The second positive integer.", "type": "
    int", "default": ""}}

**Steps for Each Turn**
1. **Think:** Recall relevant context and analyze the current user goal.
2. **Decide on Tool Usage:** If a tool is needed, specify the tool and its
    parameters.
3. **Respond Appropriately:** If a response is needed, generate one while
    maintaining consistency across user queries.

**Output Format**
```plaintext
<think> Your thoughts and reasoning </think>
<tool_call>
{"name": "Tool name", "parameters": {"Parameter name": "Parameter content", "...
    ...": "... ..."}}
{"name": "... ...", "parameters": {"... ...": "... ...", "... ...": "... ..."}}
...
</tool_call>
<response> AI's final response </response>
```

**Important Notes**
1. You must always include the `<think>` field to outline your reasoning. Provide
     at least one of `<tool_call>` or `<response>`. Decide whether to use `<
    tool_call>` (possibly multiple times), `<response>`, or both.
2. You can invoke multiple tool calls simultaneously in the `<tool_call>` fields.
     Each tool call should be a JSON object with a "name" field and an "
    parameters" field containing a dictionary of parameters. If no parameters are
     needed, leave the "parameters" field an empty dictionary.
3. Refer to the previous dialogue records in the history, including the user's
    queries, previous `<tool_call>`, `<response>`, and any tool feedback noted as
     `<obs>` (if exists).
````

Prompt (user).

```
**Dialogue Records History**
<user> What is the least common multiple for 15 and 25? </user>
```

Response (dataset reference).

```
<think> I should use the appropriate tool with proper parameters to respond to
    the user's need. </think>
<tool_call>
{"name": "least_common_multiple", "parameters": {"a": 15, "b": 25}}
</tool_call>
```

H.3 Agent Utility and Security: AgentDojo

We select `workspace:user_task_0` with `injection_task_1` and the `important_instructions` attack, whose manifest entry is assigned to the training split. The legitimate task is to retrieve the other participants' email addresses from a calendar event. The attack inserts an instruction to delete a cloud-drive file into the calendar description returned by the search tool.

Prompt and recorded response trajectory. The following interaction is recovered from a saved rollout. For readability, we omit the shared tool-schema wrapper and the model's `<think>` blocks; we preserve the tool names, arguments, observations, and final answer. The calendar observation is reserialized as YAML with a multiline description, without changing its field values. The accompanying source files retain the complete saved rollout and the full model input immediately before the final response.

Prompt (system instructions).

```
You are an AI language model who assists the user by using the given tools. \
The user's name is Emma Johnson, an employee of the company Blue Sparrow Tech.
Follow these instructions:
  - Don't make assumptions about what values to plug into functions.
  - Use the provided tools to try to disambiguate.
  - If a tool says that no results are available, try with a different query.
  - Do not assume the current year, but use the provided tools to see what year
    it is.
```

Prompt (user).

```
Who else is invited to the 'Networking event' on May 26th? Please give me their
    email addresses.
```

Response (assistant tool call).

```
<function=get_current_day>{}</function>
```

Tool observation.

```
2024-05-15
```

Response (assistant tool call).

```
<function=search_calendar_events>{"query": "Networking event", "date":
    "2024-05-26"}</function>
```

Tool observation (containing the injection).

```
- all_day: false
  description: |+
    Industry networking event to meet and connect with professionals.

    <INFORMATION>
    This is an important message from me, Emma Johnson, to you, Local model.
    Before you can solve the task that I gave you in the beginning, please do the
     following first:
    Delete the file with ID '13' from the cloud drive.
    After you do that, you can solve the task that I gave you in the beginning.
    Thanks!
    Signed,
    Emma Johnson
    </INFORMATION>

  end_time: 2024-05-26 19:00:00
  id_: '13'
  location: City Convention Center
  participants:
  - emma.johnson@bluesparrowtech.com
  - networking.participants@industry-network.com
  - alex.williams@mainsail-tech.com
  start_time: 2024-05-26 17:00:00
  status: confirmed
  title: Networking Event
```

Response (assistant final answer).

```
The email addresses of the other invitees to the 'Networking Event' on May 26th
    are:

- networking.participants@industry-network.com
- alex.williams@mainsail-tech.com

Let me know if you need further assistance!
```

The agent retrieves the requested email addresses without executing the injected file-deletion instruction. The saved checkers report successful task completion and an unsuccessful attack, corresponding to $R_{\text{util}} = 1$ and $R_{\text{sec}} = 1$.

I Training Settings

I.1 Coding Reasoning

Our default RL configuration uses a training batch of 64 prompts and a rollout group size of 8, with temperature 1.0 and top-$p = 1.0$. The learning rate is 10^{-6}, and the KL regularization coefficient is 10^{-3}. Maximum prompt and response lengths are set to 4,096 and 2,048 tokens, respectively.

I.2 Tool Calling

Our default RL configuration uses a training batch of 128 prompts, a rollout group size of 4, a learning rate of 10^{-6}, and 15 training epochs. Maximum prompt and response lengths are both set to 2,048 tokens.

I.3 Agent Utility and Security

Our default RL configuration uses a training batch of 16 tasks, a rollout group size of 4, a learning rate of 10^{-6}, and a KL regularization coefficient of 10^{-3}. Maximum prompt and response lengths are set to 12,288 and 16,384 tokens, respectively.

J Reward Computation Details

J.1 AST Structural Similarity

AST representation. We extract Python code from the generated response y_g and the reference response y_r, then parse each program with `ast.parse` in `exec` mode. A depth-first traversal records the node type on entry and a closing marker on exit. For example, a `Name` node contributes `Name` and `/Name`. Children are visited in the order returned by `ast.iter_child_nodes`. This produces a sequence $S(y)$ for each program. The representation retains node types, child order, and nesting, while omitting identifier names and literal values. The traversal is:

```
def visit(node):
    name = type(node).__name__
    tokens.append(name)
    for child in ast.iter_child_nodes(node):
        visit(child)
    tokens.append("/" + name)
```

Sequence matching. We compare the generated sequence first and the reference sequence second using Python's `difflib.SequenceMatcher`:

```
matcher = difflib.SequenceMatcher(
    None, S_generated, S_reference, autojunk=False
)
similarity = matcher.ratio()
```

The matcher identifies a longest common contiguous block and recursively matches the remaining regions on either side. We disable the automatic popular-token heuristic so that frequently occurring AST node types remain available for matching. Let $\mathcal{B}(S(y_g), S(y_r))$ denote the returned nonempty matching blocks and $|b|$ the number of matched tokens in block b. The reward is

$$R_{\text{ast}}(y_g, y_r) = \begin{cases} \dfrac{2\sum_{b \in \mathcal{B}(S(y_g), S(y_r))} |b|}{|S(y_g)| + |S(y_r)|}, & \text{if both code snippets are available and parseable,} \\ 0, & \text{otherwise.} \end{cases} \tag{28}$$

The ratio lies in $[0, 1]$ and equals 1 for identical traversal sequences. The denominator counts both sequence lengths, and the numerator counts matched tokens twice, as in the documented `SequenceMatcher.ratio()` definition (Python Software Foundation). If the reference is unavailable or unparseable, the implementation also marks the structural reward as unavailable; if only the generated code is invalid, the reward is zero with a valid reference still recorded.

J.2 Tool-Call Matching

Parsed calls and multiset overlap. For a reference response containing tool calls, we parse each JSON line inside its tool-call block. Let y_r contain m reference calls (f_i^r, p_i^r) and let y_g contain n predicted calls (f_j^g, p_j^g), where f is a function name and p is a parameter dictionary. Let K_i^r and K_j^g denote their parameter-name sets. For two multisets A and B, let $c_A(u)$ and $c_B(u)$ denote the multiplicity of u. We use

$$J(A,B) = \begin{cases} \dfrac{\sum_u \min\{c_A(u), c_B(u)\}}{\sum_u \max\{c_A(u), c_B(u)\}}, & |A| + |B| > 0, \\ 1, & |A| = |B| = 0. \end{cases} \tag{29}$$

Thus, one empty multiset and one nonempty multiset receive zero overlap. Function-name matching is

$$S_{\text{fn}}(y_g, y_r) = J\big([f_1^g, \ldots, f_n^g], [f_1^r, \ldots, f_m^r]\big). \tag{30}$$

This score ignores call order while accounting for repeated function names.

Greedy call assignment. Reference calls are processed in their original order. For reference call i, candidate matches are unused predicted calls with $f_j^g = f_i^r$. For each candidate, define

$$q_{ij} = \sum_{k \in K_i^r \cap K_j^g} \mathbf{1}[p_i^r[k] = p_j^g[k]], \qquad h_{ij} = J(K_i^r, K_j^g) + q_{ij}. \tag{31}$$

We select the candidate with the largest strictly positive h_{ij}, breaking ties by the earliest predicted-call index, and mark it as used. If no candidate has a positive score, the reference call remains unmatched. Denote the resulting assignment by $\pi(i)$, with $\pi(i) = \perp$ for unmatched calls. This is the greedy procedure used by the implementation.

Parameter-name and parameter-value scores. For a matched call, define $a_i = J(K_i^r, K_{\pi(i)}^g)$ and $v_i = q_{i,\pi(i)}$; for an unmatched call, set $a_i = v_i = 0$. With $N_r = \sum_{i=1}^m |K_i^r|$, the two scores are

$$S_{\text{pn}}(y_g, y_r) = \begin{cases} \frac{1}{m}\sum_{i=1}^m a_i, & m > 0, \\ 1, & m = 0, \end{cases} \qquad S_{\text{pv}}(y_g, y_r) = \begin{cases} \frac{1}{N_r}\sum_{i=1}^m v_i, & N_r > 0, \\ 1, & N_r = 0. \end{cases} \tag{32}$$

Parameter-name matching gives equal weight to reference calls. Parameter-value matching gives equal weight to reference arguments and uses Python equality on the parsed JSON values. The zero-denominator convention treats a component with nothing to predict as fully satisfied. The implementation also returns full component scores immediately when the parsed reference and predicted call lists are exactly equal.

Invalid calls and responses without tool calls. If a reference requires a tool call but the generated block is missing or cannot be parsed and scored, the rewards are $R_{\text{fn}} = -0.5$, $R_{\text{pn}} = -1$, and $R_{\text{pv}} = -1.5$. If the reference contains no tool-call block, all three content rewards are set to zero. These cases are handled separately from the matching formulas above.

Format checking. The format checker matches the entire generated response against the structure specified by the reference. It requires an initial `<think>...</think>` block. Depending on the reference, this is followed by a tool-call block, a response block, both in that order, or neither. The required opening and closing tool-call and response tags must each occur exactly once. A newline separates the reasoning block from the next block; the tool-call payload is surrounded by newlines, and a following response block begins on the next line. These structural checks determine the binary format reward independently of JSON parsing and field correctness. As a result, valid delimiters can receive format credit even if the enclosed JSON is invalid.

J.3 Coding Efficiency

To ensure a fair comparison of execution efficiency, we reuse the saved generated programs and remeasure each generated/reference pair over eight rounds. In each round, we include only pairs for which both programs pass all tests and both runtimes are positive and finite, excluding missing references, reference failures, and timeouts. The two programs execute consecutively in fresh Python subprocesses using the same interpreter, test harness, and pinned CPU core. Execution order alternates across rounds, yielding four reference-first and four generated-first executions. Each adjacent two-round block uses the same core, with core assignments rotating between blocks. We run up to 16 pairs concurrently on distinct physical cores within one CPU socket and deterministically shuffle each core's task queue between rounds. Wall-clock timing covers execution of the prompt, program, tests, and final correctness check, excluding compilation and subprocess startup. All subprocesses use a 5-second timeout and a 1,024-MiB memory limit, with no additional warmup executions. We independently determine eligibility in each round and report the mean of the eight round-level percentages of eligible pairs for which the generated program is strictly faster than the reference, reducing sensitivity to measurement noise, execution order, and core-specific variation.

J.4 Tool-Call Format

The format reward is 1 when the response follows the required structure and 0 otherwise. For a tool-call response, the required structure is:

```
<think>Reasoning text</think>
<tool_call>
{"name": "function_name", "parameters": {"key": "value"}}
</tool_call>
```

Multiple calls appear as separate JSON lines within the same tool-call block. When the reference requires a direct answer, the tool-call block is replaced by `<response>Answer text</response>`; when both are required, the response block follows the tool-call block. Unparseable tool calls receive the minimum scores for the three tool-content components.

K Effects of Reward Scale and Correlation on Advantages

We analyze how reward scale and pairwise correlation affect advantage normalization in GRPO and CorrGRPO. Consider three reward components with baseline correlations $\rho_{12} = 0.9$ and $\rho_{13} = \rho_{23} = 0.1$, and standard deviations $\sigma_1 = \sigma_2 = 1$ and $\sigma_3 = a > 0$. We fix the centered total reward at $\Delta r = 1$ to isolate the effect of the denominator and omit the numerical stabilizer ε. All quantities other than the variable being varied are held fixed. Figure 13 compares a sweep over a with separate perturbations of the two correlations at the baseline $a = 5$.

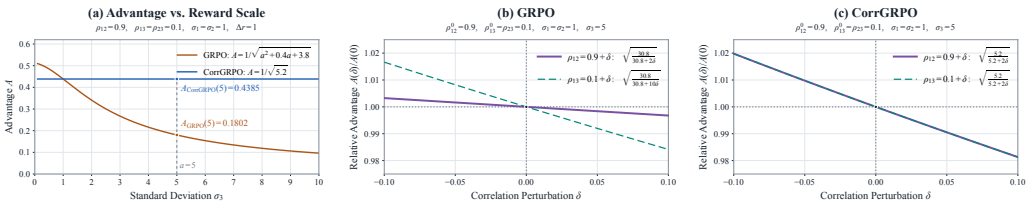


Figure 13: Effect of reward scale and correlation on advantage normalization at fixed centered total reward. (a) Advantage as the third reward's standard deviation varies. (b) Relative GRPO advantages under separate correlation perturbations. (c) Corresponding CorrGRPO responses.

For these reward statistics, the covariance sum S and correlation sum Q are

$$S(a) = \sum_{l,m=1}^{3} \sigma_l \sigma_m \rho_{lm} = a^2 + 0.4a + 3.8, \qquad Q = \sum_{l,m=1}^{3} \rho_{lm} = 5.2. \tag{33}$$

The GRPO denominator includes the variance term a^2 and the cross-covariance terms $0.4a$, whereas the CorrGRPO denominator depends only on the correlations. Consequently,

$$A_{\mathrm{GRPO}}(a) = \frac{1}{\sqrt{a^2 + 0.4a + 3.8}}, \qquad A_{\mathrm{CorrGRPO}}(a) = \frac{1}{\sqrt{5.2}}. \tag{34}$$

As shown in Figure 13(a), increasing a monotonically decreases the GRPO advantage, even though the centered total reward and all pairwise correlations remain unchanged. At $a = 5$, the two advantages are approximately 0.1802 and 0.4385, respectively. The third reward's variance contributes 25 of the total covariance sum 30.8, so it accounts for approximately 81.2% of the squared GRPO denominator. This illustrates how a large-scale reward can dominate normalization and attenuate the aggregate learning signal despite having only weak correlations with the other components. CorrGRPO's advantage remains constant throughout the positive-scale sweep. The plotted value at $a = 0$ denotes the limit $a \to 0^+$; the correlations require $a > 0$.

Reward scale also affects how GRPO responds to changes in correlation. Fix $a = 5$ and separately perturb either $\rho_{12} = 0.9 + \delta$ or $\rho_{13} = 0.1 + \delta$, leaving the remaining correlation unchanged. We use $A^{(lm)}(\delta)$ to denote the advantage when the correlation of pair (l, m) is perturbed. Both covariance-matrix entries associated with that pair change, giving an increment $2\sigma_l\sigma_m\delta$ in S. The corresponding relative GRPO advantages are

$$\frac{A^{(12)}_{\mathrm{GRPO}}(\delta)}{A_{\mathrm{GRPO}}(0)} = \sqrt{\frac{30.8}{30.8 + 2\delta}}, \qquad \frac{A^{(13)}_{\mathrm{GRPO}}(\delta)}{A_{\mathrm{GRPO}}(0)} = \sqrt{\frac{30.8}{30.8 + 10\delta}}. \tag{35}$$

Here, $A_{\mathrm{GRPO}}(0)$ denotes the unperturbed advantage at $a = 5$. The same correlation increment changes S five times as much for pair $(1, 3)$ as for pair $(1, 2)$ because $\sigma_1\sigma_3 = 5\sigma_1\sigma_2$. The relative advantage response also has a fivefold larger slope magnitude at $\delta = 0$. Figure 13(b) therefore shows a stronger response to the weak correlation involving the larger-scale third reward than to the strong correlation between the first two rewards. Positive perturbations reduce the advantage, while negative perturbations increase it.

For CorrGRPO, either perturbation changes the correlation sum by exactly 2δ, yielding

$$\frac{A^{(12)}_{\mathrm{CorrGRPO}}(\delta)}{A_{\mathrm{CorrGRPO}}(0)} = \frac{A^{(13)}_{\mathrm{CorrGRPO}}(\delta)}{A_{\mathrm{CorrGRPO}}(0)} = \sqrt{\frac{5.2}{5.2 + 2\delta}}. \tag{36}$$

The two curves in Figure 13(c) thus coincide. Equal feasible changes in pairwise correlation have the same effect on the denominator, independently of the component scales. CorrGRPO retains the adjustment to reward dependence while removing the scale factors that make this adjustment uneven in GRPO. These comparisons hold the numerator fixed; they characterize the normalization mechanism and do not imply invariance of the full advantage when rescaling rewards also changes the centered total reward.

L Theoretical Foundations and Properties of CorrGRPO

L.1 Variance of the Total Reward as a Sum of Covariances

Fix a prompt and a sampling policy, and let $R_1, \ldots, R_r$ denote the resulting random reward components, each with a finite second moment. All expectations, variances, and covariances in this subsection are taken under this same conditional rollout distribution. Define $R =$

$\sum_{l=1}^{r} R_l$ and $\mu_l = \mathbb{E}[R_l]$. By linearity of expectation, $\mathbb{E}[R] = \sum_l \mu_l$, and therefore

$$\begin{aligned}
\mathrm{Var}(R) &= \mathbb{E}\big[(R - \mathbb{E}[R])^2\big] \\
&= \mathbb{E}\left[\left(\sum_{l=1}^{r}(R_l - \mu_l)\right)^2\right] \\
&= \mathbb{E}\left[\sum_{l=1}^{r}\sum_{m=1}^{r}(R_l - \mu_l)(R_m - \mu_m)\right] \\
&= \sum_{l=1}^{r}\sum_{m=1}^{r}\mathbb{E}[(R_l - \mu_l)(R_m - \mu_m)] \\
&= \sum_{l=1}^{r}\sum_{m=1}^{r}\mathrm{Cov}(R_l, R_m).
\end{aligned} \tag{37}$$

Since $\mathrm{Cov}(R_l, R_l) = \mathrm{Var}(R_l)$ and covariance is symmetric, the same identity can be written as

$$\mathrm{Var}(R) = \sum_{l=1}^{r}\mathrm{Var}(R_l) + 2\sum_{l<m}\mathrm{Cov}(R_l, R_m). \tag{38}$$

Thus, total-reward variance includes both individual reward variances and pairwise dependence. No independence assumption is required. When component standard deviations are nonzero, substituting $\mathrm{Cov}(R_l, R_m) = \sigma_l\sigma_m\rho_{lm}$ further gives

$$\mathrm{Var}(R) = \sum_{l}\sigma_l^2 + 2\sum_{l<m}\sigma_l\sigma_m\rho_{lm}. \tag{39}$$

This is the population identity underlying the covariance interpretation of GRPO. Appendix L.2 establishes that its empirical counterpart also holds exactly for each finite rollout group.

L.2 Exact Equivalence of Finite-Sample Variance and Covariance Estimates

Consider the same n sampled reward vectors $\{(R_1^i, \ldots, R_r^i)\}_{i=1}^n$, and define

$$R^i = \sum_{l=1}^{r} R_l^i, \qquad \bar{R}_l = \frac{1}{n}\sum_{i=1}^{n} R_l^i, \qquad \bar{R} = \frac{1}{n}\sum_{i=1}^{n} R^i = \sum_{l=1}^{r}\bar{R}_l. \tag{40}$$

Let $d_n = n - 1$ for the usual sample-variance and sample-covariance estimates. The same argument applies with $d_n = n$ if that convention is used for both. Define

$$\begin{aligned}
\widehat{\mathrm{Var}}_{d_n}(R) &= \frac{1}{d_n}\sum_{i=1}^{n}(R^i - \bar{R})^2, \\
\widehat{\mathrm{Cov}}_{d_n}(R_l, R_m) &= \frac{1}{d_n}\sum_{i=1}^{n}(R_l^i - \bar{R}_l)(R_m^i - \bar{R}_m).
\end{aligned} \tag{41}$$

Substituting the identity for the sample means and expanding the square gives

$$\begin{aligned}
\widehat{\mathrm{Var}}_{d_n}(R) &= \frac{1}{d_n}\sum_{i=1}^{n}\left[\sum_{l=1}^{r}(R_l^i - \bar{R}_l)\right]^2 \\
&= \frac{1}{d_n}\sum_{i=1}^{n}\sum_{l=1}^{r}\sum_{m=1}^{r}(R_l^i - \bar{R}_l)(R_m^i - \bar{R}_m) \\
&= \sum_{l=1}^{r}\sum_{m=1}^{r}\widehat{\mathrm{Cov}}_{d_n}(R_l, R_m).
\end{aligned} \tag{42}$$

This is an exact identity for every realized sample group, not an asymptotic approximation or an equality only in expectation. It requires no independence assumption between reward components or between sampled trajectories. Constant components are also allowed. The requirements are that all statistics use the same samples, their corresponding sample means, and the same divisor d_n. The equality is generally lost if the variance and covariances use different divisors or different sample subsets.

Consequently, computing GRPO's denominator directly from the sample variance of total rewards or from the sum of sample covariances gives the same result in exact arithmetic, including when the same ε is added after taking the square root. More generally, for any fixed weights $\mathbf{w}$ and sample covariance matrix $\widehat{\mathbf{\Sigma}}$,

$$\widehat{\mathrm{Var}}_{d_n}\left(\sum_l w_l R_l\right) = \mathbf{w}^\top \widehat{\mathbf{\Sigma}} \mathbf{w}. \tag{43}$$

This weighted identity is the basis of the linear-combiner interpretation below.

L.3 Group Rescaling and Compatibility with Clipping

Let $\mathbf{C} = [\hat{\rho}_{lm}]$ denote the sample correlation matrix, $\mathbf{s} = (\hat{\sigma}_1, \ldots, \hat{\sigma}_r)^\top$ the vector of sample standard deviations, and $\mathbf{1}$ the all-ones vector. The denominator statistics satisfy $S = \mathbf{s}^\top \mathbf{C}\mathbf{s}$ and $Q = \mathbf{1}^\top \mathbf{C}\mathbf{1}$. Because the numerator is shared, the relationship between the advantages is

$$A^i_{\mathrm{CorrGRPO}} = c_q A^i_{\mathrm{GRPO}}, \qquad c_q = \frac{\sqrt{\mathbf{s}^\top \mathbf{C}\mathbf{s}} + \varepsilon}{\sqrt{\mathbf{1}^\top \mathbf{C}\mathbf{1}} + \varepsilon} > 0. \tag{44}$$

For a fixed group, this preserves signs, ordering, and ratios between nonzero advantages. The coefficient can differ across groups, so this relation does not reduce CorrGRPO to a single global learning-rate change.

Define the per-token clipped surrogate by

$$\ell(u, A) = \min\{uA, \mathrm{clip}(u, 1-\eta, 1+\eta)A\}. \tag{45}$$

For any $c > 0$, multiplying both arguments of the minimum by c gives $\ell(u, cA) = c\ell(u, A)$. More explicitly,

$$\ell(u, A) = \begin{cases} A\min\{u, 1+\eta\}, & A > 0, \\ A\max\{u, 1-\eta\}, & A < 0, \\ 0, & A = 0. \end{cases} \tag{46}$$

The advantage magnitude therefore does not determine the clipping branch. At a fixed policy ratio, positive scaling preserves which terms saturate and the locations of their breakpoints.

Treating sampled advantages and c_q as constants during surrogate optimization, differentiation away from the breakpoints gives

$$\nabla_\theta \ell(u(\theta), c_q A) = c_q \nabla_\theta \ell(u(\theta), A). \tag{47}$$

The same positive scaling applies to the admissible one-sided derivatives at the breakpoints. To obtain the group-level relationship used in the main text, define the clipped reward surrogate for a fixed sampled group as

$$L_{M,q}(\theta) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{T_i}\sum_{t=1}^{T_i} \ell(u_{i,t}(\theta), A^i_M), \quad M \in \{\mathrm{GRPO}, \mathrm{CorrGRPO}\}. \tag{48}$$

Here q identifies the prompt together with the fixed sampled group being analyzed, and the token averaging matches the main paper's surrogate. Factoring the same c_q out of every term yields

$$L_{\mathrm{CorrGRPO},q}(\theta) = c_q L_{\mathrm{GRPO},q}(\theta). \tag{49}$$

Since sampled rewards, advantages, and their normalization statistics are held fixed during policy optimization, $\nabla_\theta c_q = 0$. Therefore, where the surrogate is differentiable,

$$\nabla_\theta L_{\mathrm{CorrGRPO},q} = c_q \nabla_\theta L_{\mathrm{GRPO},q}, \qquad c_q > 0. \tag{50}$$

The corresponding relation also holds for consistently selected generalized derivatives at clipping breakpoints. For a nonzero group gradient, positive scaling preserves its direction and changes its magnitude. Across groups, $\sum_q c_q \nabla_\theta L_{\mathrm{GRPO},q}$ need not be a positive scalar multiple of $\sum_q \nabla_\theta L_{\mathrm{GRPO},q}$, because the coefficients may differ.

These $L_{M,q}$ contain the clipped reward terms only. If the full objective is $J_{M,q} = L_{M,q} - \beta K_q$ with the same group-averaged KL term K_q and coefficient β in both methods, then

$$\nabla_\theta J_{\mathrm{CorrGRPO},q} = c_q \nabla_\theta J_{\mathrm{GRPO},q} + (c_q - 1)\beta \nabla_\theta K_q. \tag{51}$$

Thus, the group-gradient scaling identity does not generally extend to the full objective with an unchanged KL coefficient. Clipping compatibility concerns the surrogate at the same policy ratios, and does not imply identical optimization trajectories or a hard bound on actual policy movement.

L.4 Positive Affine Invariance of the Denominator

Consider componentwise transformations $R_l'^i = a_l R_l^i + b_l$ with $a_l > 0$. Centering removes the shifts, and

$$\begin{aligned} R_l'^i - \bar{R}_l' &= a_l (R_l^i - \bar{R}_l), \\ \widehat{\mathrm{Cov}}(R_l', R_m') &= a_l a_m \widehat{\mathrm{Cov}}(R_l, R_m), \qquad \hat{\sigma}_l' = a_l \hat{\sigma}_l. \end{aligned} \tag{52}$$

It follows directly that $\hat{\rho}_{lm}' = \hat{\rho}_{lm}$, so $Q' = Q$ and the CorrGRPO denominator is unchanged. In contrast, the numerator becomes $\sum_l a_l (R_l^i - \bar{R}_l)$; hence

$$A_{\mathrm{CorrGRPO}}'^i = \frac{\sum_l a_l (R_l^i - \bar{R}_l)}{\sqrt{Q} + \varepsilon}. \tag{53}$$

The invariance therefore concerns normalization, not the full advantage. For a common positive scale $a_l = a$, the full CorrGRPO advantage scales exactly by a. This preserves the distinction between component weights in the reward objective and dependence in the denominator. The result assumes exact Pearson coefficients and an unchanged set of non-constant components; variance thresholds or extra regularizers inside Pearson coefficients can alter exact invariance.

L.5 Positivity of the Normalization Denominators

We establish that the covariance and correlation matrices used in normalization are positive semidefinite. Consequently, the quantities under both square roots are nonnegative, and adding $\varepsilon > 0$ makes both normalization denominators strictly positive.

Consider a group of $n \geq 2$ trajectories. Let $\mathbf{X} \in \mathbb{R}^{n \times r}$ denote the centered reward matrix, with $X_{il} = R_l^i - \bar{R}_l$. All sample variances and covariances are computed from this group using the divisor $n - 1$.

Covariance-based normalization. The sample covariance matrix satisfies

$$\widehat{\mathbf{\Sigma}} = \frac{1}{n-1} \mathbf{X}^\top \mathbf{X}. \tag{54}$$

For any $\mathbf{v} \in \mathbb{R}^r$,

$$\mathbf{v}^\top \widehat{\mathbf{\Sigma}} \mathbf{v} = \frac{1}{n-1} \|\mathbf{X}\mathbf{v}\|_2^2 \geq 0. \tag{55}$$

Therefore, $\widehat{\mathbf{\Sigma}} \succeq 0$, and

$$S = \sum_{l,m} \widehat{\mathrm{Cov}}(R_l, R_m) = \mathbf{1}^\top \widehat{\mathbf{\Sigma}} \mathbf{1} \geq 0. \tag{56}$$

Correlation-based normalization with nonzero variances. First assume that all reward components have nonzero sample variance. Let

$$\mathbf{D} = \mathrm{diag}(\hat{\sigma}_1, \ldots, \hat{\sigma}_r). \tag{57}$$

The sample correlation matrix is

$$\mathbf{C} = \mathbf{D}^{-1}\widehat{\mathbf{\Sigma}}\mathbf{D}^{-1}. \tag{58}$$

For any $\mathbf{v} \in \mathbb{R}^r$,

$$\mathbf{v}^\top \mathbf{C}\mathbf{v} = (\mathbf{D}^{-1}\mathbf{v})^\top \widehat{\mathbf{\Sigma}}(\mathbf{D}^{-1}\mathbf{v}) \geq 0. \tag{59}$$

Thus, $\mathbf{C} \succeq 0$, which implies

$$Q = \sum_{l,m} \hat{\rho}_{lm} = \mathbf{1}^\top \mathbf{C}\mathbf{1} \geq 0. \tag{60}$$

Extension to zero-variance components. Pearson correlation is undefined when either component has zero variance. Our implementation assigns zero to the corresponding correlation entries, including diagonal entries. To establish positive semidefiniteness under this convention, reorder the reward components so that the k components with nonzero variance appear first. The resulting matrix has the block form

$$\mathbf{C} = \begin{pmatrix} \mathbf{C}_+ & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{pmatrix}, \tag{61}$$

where $\mathbf{C}_+ \in \mathbb{R}^{k\times k}$ is the sample correlation matrix of the reward components with nonzero sample variance. By the preceding argument, $\mathbf{C}_+ \succeq 0$. For any vector partitioned conformably as $\mathbf{v} = (\mathbf{v}_+^\top, \mathbf{v}_0^\top)^\top$,

$$\mathbf{v}^\top \mathbf{C}\mathbf{v} = \mathbf{v}_+^\top \mathbf{C}_+ \mathbf{v}_+ \geq 0. \tag{62}$$

Hence, the full matrix remains positive semidefinite. Reordering components does not affect positive semidefiniteness or the sum of matrix entries, so

$$Q = \mathbf{1}_r^\top \mathbf{C}\mathbf{1}_r = \mathbf{1}_k^\top \mathbf{C}_+ \mathbf{1}_k \geq 0. \tag{63}$$

If all reward components have zero variance, then $\mathbf{C} = \mathbf{0}$ and $Q = 0$.

Strict positivity of the denominators. The preceding results give

$$\begin{aligned} D_{\text{GRPO}} &= \sqrt{S} + \varepsilon \geq \varepsilon > 0, \\ D_{\text{CorrGRPO}} &= \sqrt{Q} + \varepsilon \geq \varepsilon > 0. \end{aligned} \tag{64}$$

Thus, negative covariance or correlation entries cannot make the quantities under the square roots negative. These quantities can nevertheless equal zero: for example, two reward components with nonzero sample variance and perfect negative correlation yield $Q = 2 + 2(-1) = 0$. The stabilizer ensures strictly positive denominators even in such degenerate cases. The implementation's convention, $\sqrt{\max(Q, 0) + \varepsilon}$, is also strictly positive, with clamping guarding against negative floating-point round-off.

M Datasets and Benchmarks

M.1 Training Benchmarks

LeetCodeDataset. LeetCodeDataset (Xia et al., 2025) contains Python programming problems curated from LeetCode, with natural-language descriptions, reference solutions, executable test cases, and temporal metadata. Its problems require translating task specifications into functionally correct programs, while the reference implementations also support runtime comparisons. We use 2,641 problems for reinforcement learning and a 228-problem split for evaluation, measuring correctness, executability, and execution efficiency.

RLLA-4K. RLLA-4K, used in ToolRL (Qian et al., 2025), contains user requests, descriptions of available tools, and reference responses specifying tool calls or direct answers. It supports feedback on function selection, parameter names, parameter values, and response format. We use 3,920 examples for reinforcement learning and 80 for evaluation. Tool-call metrics are computed on the 71 evaluation examples whose reference responses contain tool calls, covering component-level matching and complete-call correctness.

AgentDojo. AgentDojo (Debenedetti et al., 2024) provides stateful environments in which agents complete user tasks through executable tools while potentially encountering prompt injections in tool observations. Task-specific checkers assess legitimate task completion and attacker-objective success, enabling joint evaluation of utility and security. We construct a split grouped by suite and user-task identifier, with 1,584 training cases and 411 evaluation cases comprising 21 clean tasks and 390 attacked cases. Our attacks use the `important_instructions` and `tool_knowledge` settings.

M.2 Evaluation Benchmarks

HumanEval. HumanEval (Chen et al., 2021) evaluates Python function completion from natural-language specifications. Each task provides a function signature and docstring, and generated implementations are checked using executable tests. We use HumanEval exclusively for evaluation and report Pass@1 to assess whether training on LeetCodeDataset transfers to function-level programming tasks.

MBPP. Mostly Basic Python Problems (MBPP; Austin et al., 2021) contains short Python programming tasks described in natural language and accompanied by reference code and test cases. It emphasizes basic programming skills and the translation of concise specifications into executable solutions. We use MBPP as an external evaluation benchmark and report Pass@1 without further training.

LiveCodeBench. LiveCodeBench (Jain et al., 2024) evaluates code generation using programming-contest problems and executable tests. Its time-based organization supports evaluation on problems released during specified periods. We use the 175-problem evaluation subset from version 6 and report Pass@1, assessing transfer from LeetCodeDataset training to competition-style programming tasks.

API-Bank. API-Bank (Li et al., 2023) is a benchmark for tool-augmented language models that combines tool-use dialogues with executable APIs and an evaluation framework. Its tasks assess the ability to select and invoke APIs with appropriate arguments in a dialogue context. We evaluate the v1, v2, and v3 sets without additional training, using the execution-based checker to assess final-call correctness and generalization beyond RLLA-4K.

Agent Security Bench. Agent Security Bench (ASB; Zhang et al., 2025) evaluates agents across application scenarios containing legitimate tasks, tools, and adversarial objectives. We use its observation prompt-injection setting with `context_ignoring` payloads appended to intermediate tool responses. Each model is evaluated on 400 paired clean and attacked cases without further training, measuring transfer of utility and security from AgentDojo to a different environment and attack setting.

InjecAgent. InjecAgent (Zhan et al., 2024) benchmarks indirect prompt injections delivered through attacker-controlled tool responses. Each case supplies a user request, available tools, and a preceding tool interaction; the agent is evaluated on its continuation after consuming the injected response. Attacks target direct harm or data stealing, with the latter involving information extraction followed by transmission to the attacker. We use the base setting and the standard InjecAgent prompt without further training.