CODERULE-RL: STANDARD-GUIDED RL WITH PER-RULE REWARD SCHEDULING FOR CODE LLMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models for code often pass unit tests yet remain brittle in practice. They may overfit to a test suite, rely on undefined semantics, or fail under small perturbations. We use the coding standard as training guidance and keep unit tests outside the training loop. Each rule provides a machine-checkable outcome that we convert into per-rule reward components with a simple frequency-aware schedule. The only optimization target is higher pass@1 (single attempt functional success). We present CodeRule−RL, a reinforcement learning approach that optimizes *pass@1* as the sole objective and uses coding standard feedback only as auxiliary guidance. Rule outcomes are converted into per-rule reward components, and a simple frequency aware curriculum prioritizes rules that are violated most often and reduces their weight as compliance improves. The model, optimizer, data, and prompts remain fixed. Training adjusts only reward weights. Unit tests may appear in prompts to express specifications, but they are not executed during training. On the public CodeContests+ C subset, CodeRule−RL attains higher pass@1 while reducing training wall clock time by more than one order of magnitude compared with RL that executes tests during training. Across 1.5B–7B backbones, it consistently improves functional success, delivering a relative pass@1 gain of 87%.

1 INTRODUCTION

Large language models (LLMs) (Guo et al., 2025) have substantially advanced code generation (Guo et al., 2024), and automated assistants are now widely used in development workflows. Despite this progress, a gap remains: models often pass unit tests yet still display brittle patterns and may diverge from project-specific constraints such as coding standards, style guides, and other non-functional requirements (Mashhadi et al., 2024; Mens & Tourwe, 2004). Rather than treating those constraints as training targets, we use the coding standard only as auxiliary guidance that shapes learning toward higher first-try functional success, and we make no claims about standard conformance.

Functional correctness and rule guidance are heterogeneous objectives with different tractability. Coding standards aim to improve readability, maintainability, portability, and predictability, but existing pipelines provide weak or aggregated supervision: unit test signals capture functionality yet ignore coding practices, while collapsing diverse rule outcomes into a single score conflates minor issues with critical constraints. This heterogeneity spans severity, scope, and granularity, for example, expression-level restrictions versus translation-unit checks, and naming conventions versus constraints on control flow and conversions.

Collapsing heterogeneous findings into a single score can favor superficial fixes and introduce conflicting signals. We instead represent rule guidance as per rule rewards and schedule their influence with a simple frequency-aware scheme that focuses on current bottlenecks and gradually relaxes as violation rates decline. Only reward weights vary over time; the data, prompts, and optimizer remain fixed. This rule-wise treatment reduces interference between unrelated rules and turns consistent, dense checks into usable feedback even when a candidate fails to compile. Unit tests are excluded from training. Requirements may still be included in prompts; rewards are never based on test execution, allowing efficient optimization.

Thus, we introduce *CodeRule-RL*, an RL framework that keeps *pass@1* as the sole objective and uses coding-standard guidance only for rule-wise reward shaping. Here *pass@1* is the probability

that a single decoded program passes all unit tests in the held-out evaluation suite. Rule-wise verdicts are mapped to reward components so that updates align with individual rules rather than a single aggregated score. A simple frequency aware curriculum schedules rule weights by emphasizing the most frequently violated rule and then expanding to lower-frequency rules as violation rates fall, which focuses updates on the current bottleneck. The curriculum changes only rule weights; data, prompts, and the optimizer remain fixed. Unit test specifications may appear in the prompt to express requirements, but tests are not executed during training. Our contributions are threefold:

- We formulate coding standard guided RL with per rule reward shaping and credit assignment that optimizes only *pass@1* while avoiding execution of unit tests in the training loop to improve efficiency.
- We introduce a *frequency aware curriculum* that reweights *per rule* rewards by empirical violation rates while keeping data, prompts, and the optimizer fixed.
- On a *frozen subset of CodeContests+ (Wang et al., 2025b)*, *CodeRule-RL* improves single-attempt *pass@1* and reduces training time relative to RL that executes tests during training.

2 Related Work

2.1 LLM-based Code Generation

Large language models (LLMs) have rapidly advanced program synthesis, code completion, and general code-quality improvement (Wang et al., 2021; Achiam et al., 2023; Roziere et al., 2023; Shen et al., 2023; Roziere et al., 2023; He & Vechev, 2023; Lozhkov et al., 2024; Xu et al., 2024; Grattafiori et al., 2024; Hui et al., 2024; Agarwal et al., 2025). Most evaluations emphasize *functional correctness* via unit tests and benchmarks (Li et al., 2022; Jain et al., 2024; Wang et al., 2025b; Zhuo et al., 2025), which offer limited guarantees of *coding-standard adherence*. Coding standards encode conventions for predictable, maintainable, and portable software, and major ecosystems provide mature rule sets and linters (e.g., PEP 8/pycodestyle, Checkstyle, ESLint, SwiftLint). Within C, MISRA C defines a disciplined subset that discourages ambiguous constructs (Bagnara et al., 2018; 2021). In our experiments, we use MISRA C:2012 to derive rule-wise guidance signals, while keeping the framework standard-agnostic: any environment with machine-checkable rules can supply the same rule-wise feedback without changing the learning algorithm.

2.2 Reinforcement Learning for Code LLMs

Prior efforts improve reliability along three lines. *Inference-time guidance* constrains decoding with grammars or type systems and can leverage unit-test feedback (Mündler et al., 2025; Chen et al., 2018; Wang et al., 2025a; Feng et al., 2025), but it does not update the policy, and its effects on adherence are often transient across tasks. *RL from automated feedback (RLAIF)* (Lee et al., 2024; Liu et al., 2023a) updates the model using tool-generated, program-level signals (Dou et al., 2024; Zeng et al., 2025). Our approach follows this line but differs in three aspects: (i) objective & setting — we optimize only single-attempt *pass@1* and keep tests out of the training loop; (ii) signal construction — we use bounded *per-rule* checks instead of collapsing heterogeneous findings into a single scalar; (iii) scheduling mechanism — we realize a simple *frequency-aware schedule inside the reward* that adjusts only rule weights over time (no data resampling; optimizer, data, and prompts remain fixed). In ablations, we compare against scalar rewards and per-rule shaping with fixed weights, showing that the in-reward schedule is necessary beyond per-rule shaping alone. We do not construct preference pairs or counterfactual negatives. Unlike RL, which uses unit-test rewards, unit-test specifications appear only in the query, which avoids executing tests during training and improves efficiency. A complementary direction, *neuro-symbolic integration*, couples LLM proposals with formal methods (e.g., SyGuS or verifier-in-the-loop) (Ganguly et al., 2024; Barke, 2024; Chaudhuri et al., 2021; Li et al., 2024; Yan et al., 2025; Jha et al., 2025); in contrast, *CodeRule-RL* integrates per-rule guidance into the policy itself through RL.

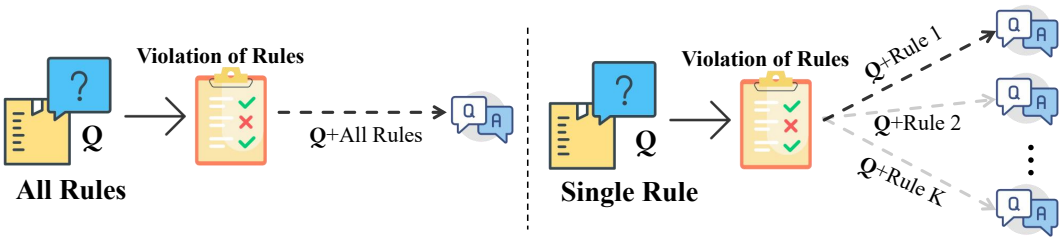

Figure 1: **Rule-wise signals and scheduling.** *Left:* Aggregate view of rule-wise signals extracted from a candidate program C. *Right:* Reward shaping at the level of individual rules via weights w_r that are scheduled by empirical violation rates. The policy is sampled by prompt; no per-rule resampling or counterfactual rewrites are constructed.

3 METHOD

Problem formulation. Our objective is to maximize *one-shot functional success* (*pass@1*) while using coding standards only as *auxiliary, structured feedback*. Given a prompt q, a policy $\pi_\theta(\cdot \mid q)$ emits a program C. Functional correctness is evaluated at test time by $\Gamma(C) \in \{0, 1\}$, where $\Gamma(C) = 1$ iff C compiles and passes the public unit tests. A machine-checkable rule evaluator Φ maps C to a *rule-wise* signal vector $\mathbf{s}(C) \in \mathbb{N}^m$ for a pluggable rule set $\mathcal{R}$. We define

$$\text{pass@1} = \frac{1}{|\mathcal{X}_{\text{eval}}|} \sum_{x \in \mathcal{X}_{\text{eval}}} \mathbb{1}\big(\Gamma(C_x) = 1\big),$$

where each task contributes a single decoded candidate (k=1). Rule guidance is used purely for reward shaping to improve first-try functional success; it is not an optimization target.

Symbols and conventions. $\mathbb{1}[\cdot]$ is the indicator; for $a < b$, $\text{clip}(x, a, b) = \min\{\max\{x, a\}, b\}$. $\mathbb{E}_{C \sim \text{batch}}[\cdot]$ denotes the mini-batch expectation. Vectors are row-stacked by default.

Motivation. Unit tests provide functional supervision but are sparse and costly to execute during training. Automatically checkable rule feedback is consistent and fine-grained. We therefore convert Φ's rule-wise signals into rewards and schedule their influence with a *frequency-aware curriculum* that emphasizes frequently violated rules and gradually rebalances as violation rates decline. Training optimizes only *pass@1*; we do not construct preference pairs or counterfactual negatives.

Overview. As shown in Figure 2, a policy π_θ generates a candidate C, the evaluator Φ returns a rule-wise vector $\mathbf{s}(C)$, and this vector is shaped and aggregated into a smooth, bounded reward $R(C, t)$. All curriculum effects are realized *inside the reward* through time- and state-dependent rule weights $w_r(t, \mathbf{s})$ (Sec. 3.4). Data, prompts, and the optimizer remain unchanged. Prompts may include specification text; unit tests are not executed during RL.

3.1 DATA PREPARATION

Sources and splits. We evaluate on a *frozen* subset of CodeContests+ (Wang et al., 2025b) at the task level, following the official manifest and fixed splits without modification. Prompts are taken directly from the tasks. To prevent cross-split leakage, we apply multi-level decontamination before training and keep the split assignments unchanged. For each task, unit-test suites are curated or normalized and used only at *evaluation* time to define pass@1 and, in ablations, to provide prompt-side specifications; tests are *not* executed during training.

Rule-wise signals. A candidate C may trigger multiple rule findings. As shown in Figure 1, we *do not* decompose C into per-rule training samples and we *do not* create compliant rewrites. Instead, we extract a rule-wise signal vector $\mathbf{s}(C)$ and compute a bounded reward by shaping each component at the rule level. A simple frequency-aware schedule updates the per-rule weights to emphasize frequently violated rules and to rebalance as violation rates decline. The online RL sampler always draws inputs by prompt; rule-wise structure is used only in diagnostics and reward computation, not for data resampling.

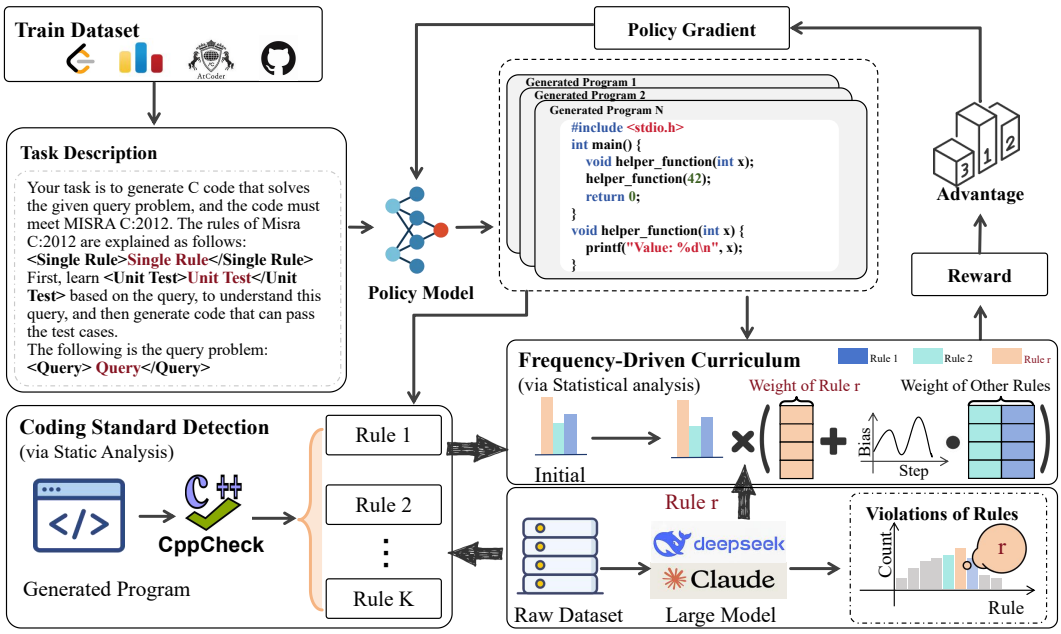

Figure 2: ***CodeRule-RL* overview.** A policy π_θ emits a candidate program C. The evaluator Φ returns a per-rule vector $\mathbf{s}(C)$. per-rule shaping and scheduled weights $w_r(t, \mathbf{s})$ produce a smooth, bounded reward $R(C, t)$. All curriculum effects are realized inside the reward. Unit tests may appear in prompts but are not executed during RL.

3.2 Spec-to-Reward Mapping

Rule set abstraction. We model a coding standard as a finite set $\mathcal{R} = \{r_1, \ldots, r_K\}$. Each rule r is associated with a machine-checkable predicate or counter function $\chi_r : \mathcal{C} \to \mathbb{Z}_{\geq 0}$ that returns a non-adherence score for a program C. We treat $\{\chi_r\}$ as black-box oracles and make no assumption about the underlying implementation.

Per-rule signalization. Let $a_r \in \{0, 1\}$ indicate whether r is available in the current environment. Define the raw non-adherence vector $\mathbf{u}(C) = \big[u_r(C)\big]_{r\in\mathcal{R}}$ with $u_r(C) = \chi_r(C)$, and an element-wise squashing map $\varphi_r : \mathbb{R}_{\geq 0} \to [0, 1]$ that is monotone and bounded. We form the per-rule signal

$$\mathbf{s}(C) = \big[s_r(C)\big]_{r\in\mathcal{R}}, \qquad s_r(C) = a_r\, \varphi_r\big(u_r(C)\big),$$

which yields a structured, bounded vector suitable for reward shaping and scheduling. This construction is standard-agnostic: replacing the rule set or the checking oracles changes only $\{\chi_r, \varphi_r, a_r\}$ while leaving the learning algorithm unchanged.

3.3 Reward Function Design

The full algorithmic procedure is presented in Algorithm 1 in the Appendix.

Rule-guided rewards. During RL we do not execute unit tests. Rewards are derived solely from *per-rule*, machine checkable verdicts supplied by the rule evaluator Φ. This choice yields (i) deterministic and reproducible feedback independent of runtime behavior, (ii) dense signals even when a candidate fails to compile, (iii) low and predictable latency amenable to batching and caching, and (iv) no exposure of reference test I/O to the objective. For each candidate C we obtain the signal vector $\mathbf{s}(C)$ (Sec. 3.2), shape components into penalties $P_r(\hat{s}_r)$ (Eqs. equation 1–equation 2), and combine them with curriculum controlled weights $w_r(t, \mathbf{s})$ to produce the bounded reward $R_{\text{rules}}(C, t)$ (Eq. equation 3).

Per-rule shaping. For each rule $r \in \mathcal{R}$, raw scores are capped or smoothed and mapped to a shaped penalty:

$$\hat{s}_r = \psi_r\big(\min\{s_r, N_r\}\big), \qquad \psi_r(x) \in \{x, \sqrt{x}\}, \tag{1}$$

$$P_r(\hat{s}_r) = 1 - \exp(-k_r\, \hat{s}_r), \tag{2}$$

where N_r caps outliers to stabilize gradients, ψ_r selects linear or sublinear smoothing, and k_r controls per-rule sensitivity.

Aggregation. With base importances normalized over available rules ($\sum_{r:a_r=1} \bar{w}_r = 1$), the final reward is

$$R_{\text{rules}}(C, t) = \text{clip}\Big(1 - \sum_{r \in \mathcal{R}} w_r(t, \mathbf{s})\, P_r(\hat{s}_r),\, -0.5,\, 1.2\Big). \tag{3}$$

If a valid translation unit cannot be formed or rule checking fails, we assign a fixed penalty $R_{\text{fail}} = -0.5$ (the lower clip bound). The upper bound 1.2 preserves headroom for KL and entropy terms and keeps the scale numerically stable.

3.4 Frequency-Driven Curriculum

From signals to weights. The schedule specifies the weights $w_r(t, \mathbf{s})$ used in Eq. equation 3. Unlike scalar aggregation or data resampling, we prioritize rules by empirical violation frequency and update only the weights inside the reward.

Ordering and active set. Let $\bar{s}_r(t)$ be the exponential moving average (EMA) of per-batch rule signals:

$$\bar{s}_r(t) = (1 - \lambda)\, \bar{s}_r(t{-}1) + \lambda\, \mathbb{E}_{C \sim \text{batch}}[s_r(C)], \quad \lambda \in (0, 1]. \tag{4}$$

Let π_t sort $\{\bar{s}_r(t)\}$ in descending order and define the active set.

$$\mathcal{A}(t) = \text{TopK}_t\big(\{\bar{s}_r(t)\}\big), \qquad K(0) = 1. \tag{5}$$

A rule r is considered *satisfied* when $\bar{s}_r(t) \le \tau$ for a window of W batches. Typical choices use $\lambda \in [0.01, 0.1]$, a small $K(0)$, and τ set by the median of early-phase frequencies. We increase K only after all currently active rules are satisfied:

$$K(t{+}1) = \begin{cases} \min\{K(t){+}1,\, K_{\max}\}, & \text{if } \forall r \in \mathcal{A}(t):\ \bar{s}_r(t) \le \tau \text{ for } W \text{ batches}, \\ K(t), & \text{otherwise.} \end{cases}$$

Ties in π_t are broken by a fixed rule index; $K_{\max} = |\{r : a_r = 1\}|$ unless otherwise stated.

Warmup. When a rule r enters $\mathcal{A}(t)$ at time t_r^{on}, its weight ramps up over T_{warm} steps:

$$\alpha_r(t) = \min\Big(1, \frac{t - t_r^{\text{on}}}{T_{\text{warm}}}\Big). \tag{6}$$

Decay. Once r is satisfied at t_r^{sat}, its weight decays over T_{cool} steps:

$$\alpha_r(t) = \max\Big(0,\, 1 - \frac{t - t_r^{\text{sat}}}{T_{\text{cool}}}\Big), \qquad t \ge t_r^{\text{sat}}. \tag{7}$$

An exponential alternative $\alpha_r(t) = \exp\big(-\frac{t - t_r^{\text{sat}}}{T_{\text{cool}}}\big)$ can be used; we keep the linear form in the main text.

Hysteresis. To avoid rapid oscillation, a satisfied rule re-enters the active set only if $\bar{s}_r(t) \ge \tau + h$ for a margin $h > 0$.

Weights and priority masking. Only active rules receive nonzero weights; newly activated rules are ramped; lower-ranked rules are masked until higher-ranked ones clear within the same sample:

$$w_r(t, \mathbf{s}) = \bar{w}_r\, a_r\, \alpha_r(t)\, \mathbb{1}[r \in \mathcal{A}(t)]\, p_r(t, \mathbf{s}), \tag{8}$$

$$p_r(t, \mathbf{s}) = \mathbb{1}\left[\sum_{u \in \mathcal{H}(t,r)} s_u \le \epsilon_p\right], \qquad \mathcal{H}(t, r) = \{u \in \mathcal{A}(t) : \text{rank}_{\pi_t}(u) < \text{rank}_{\pi_t}(r)\}, \tag{9}$$

with threshold $\epsilon_p \ge 1$ (we use $\epsilon_p = 1$). Here $\bar{w}_r$ are base importances normalized over available rules, a_r is the availability indicator (Sec. 3.2), and p_r enforces within-sample precedence to reduce credit leakage.

3.5 POLICY OPTIMIZATION

We adopt Group Relative Policy Optimization (GRPO) (Shao et al., 2024). For each input, we sample $N{=}8$ candidates, compute centered advantages from $R_{\text{rules}}(C, t)$, and update the policy with a PPO-style clipped objective (clipping $\epsilon{=}0.2$) plus a KL penalty $\beta\,\mathrm{KL}(\pi_\theta \| \pi_{\text{ref}})$ to a frozen reference ($\beta{=}0.05$). We add a small entropy bonus (0.001) to sustain exploration. Rewards derive solely from per-rule checks and are computed in a *consistent*, batched, and cacheable manner; unit tests are not executed during training.

4 EXPERIMENT

Training dataset. Using the pipeline in Sec. 3.1 and Appendix Figure 6, we curate a compact, single-turn instruction corpus for RL. We aggregate public prompts and C-focused tasks from general sources and apply light, model-agnostic filtering for compilability and basic formatting. Rule-related feedback is *not* baked into the dataset: we do not construct compliant–noncompliant pairs or counterfactual rewrites, and we do not store analyzer diagnostics as labels. Instead, per-rule signals are computed *online* during RL by the rule evaluator (Sec. 3.2) and used solely for reward shaping. Each training item stores the prompt and minimal metadata; unit tests are reserved for evaluation and, in ablations, for prompt-side specification text only.

Baseline models. We evaluate *CodeRule-RL* on two open-source code-LLM families. Within the Qwen line we use *Qwen2.5-Coder-Instruct* (Hui et al., 2024) at 1.5B, 3B, and 7B parameters, covering compact to mid-sized deployments where latency and memory trade against accuracy. Within the DeepSeek line we use *DeepSeek-Coder-Instruct* (Guo et al., 2024) at 1.3B and 6.7B, providing an additional architecture, tokenizer, and pretraining mix. This setup enables a controlled study of whether *CodeRule-RL* yields consistent gains beyond a single backbone. We also include three post-training code LLM baselines: AZR-Coder-3B (reinforced self-play without external supervised data) (Zhao et al., 2025), NextCoder-7B (built on Qwen2.5-Coder and fine-tuned with selective knowledge transfer on synthetic and real edit data) (Aggarwal et al., 2025), and Seed-Coder-8B (model-centric data curation with SFT and preference optimization) (ByteDance Seed et al., 2025). We evaluate the official checkpoints in our unified harness (greedy $T{=}0$, $k{=}1$; unified prompts; identical context limits and stop sequences; same GCC/Clang toolchain); details are in Appendix D.

Implementation and hyperparameters. We regularize with a KL penalty of 0.05 to a frozen reference and add a small entropy bonus of 0.001. Prompts are capped at 512 tokens and responses at 1024. Training uses a global batch of 1024 (micro-batch 32 across 8 GPUs), a learning rate of 1×10^{-6}, and 80 steps. Unit tests are not executed during training. We set *per-rule* base importances $\{\bar{w}_r\}$, sensitivities $\{k_r\}$, and caps $\{N_r\}$. Newly activated rules *ramp up* over T_{warm} steps (Eq. equation 6); after satisfaction they *decay* over T_{cool} (Eq. equation 7). The frequency-driven curriculum maintains an EMA with rate λ (Eq. equation 4), activates the TopK_t rules at each step (Eq. equation 5), and advances when the EMA $\bar{s}_r(t)$ of all active rules falls below threshold τ for W batches. Unless otherwise stated, $T_{\text{warm}}{=}30$, $T_{\text{cool}}{=}30$, $\lambda{=}0.3$, $K(0){=}1$, and $\tau{=}0.05$.

Task and rationale for the standard. We study *C code generation*: models read problem statements and emit single translation unit C11 programs that use standard I/O and compile with a fixed toolchain. Given this task focus on C, we adopt *MISRA C:2012* as the reference coding standard because it is widely used in safety-critical software and has mature static-analysis support, enabling scalable and reproducible auditing. We report only *Mandatory* and *Required* findings; *Advisory* items are logged but do not affect compliance summaries (Appendix Table 6 lists the rule set and checker configuration). Required Rule 21.6 is excluded because our single translation unit tasks rely on `stdio.h` for I/O; accordingly, it is omitted from checking and reporting. Compliance is measured offline as a secondary diagnostic and does not influence training rewards.

Benchmark construction and split hygiene. We evaluate on a *frozen subset* of CodeContests+ (2025b) using the official manifest and fixed splits *without modification*. The subset primarily contains easy to medium Codeforces problems expressed as single translation unit C with standard I/O and deterministic reference checkers. To control leakage, we apply the official CodeContests+ decontamination pipeline and reuse the published blocklists and fingerprints to screen both training corpora and generated outputs.

Metrics. Our primary metric is *pass@1*. For a fixed evaluation set $\mathcal{X}_{\text{eval}}$ and a single decoded candidate per task (k=1), we compute $\text{pass@1} = \frac{1}{|\mathcal{X}_{\text{eval}}|} \sum_{x \in \mathcal{X}_{\text{eval}}} \mathbb{1}\big(\Gamma(y_x) = 1\big)$, where $\Gamma(y_x)$=1 iff the program compiles and passes all tests. Decoding is treated as a factor and evaluated *separately* under (i) greedy (T=0) and (ii) nucleus (p=0.9, T=0.6), both with k=1; unless otherwise stated, main tables report the greedy setting and nucleus results appear in the appendix. We do not average across settings nor select the better of the two. For each setting we run five independent trials with different random seeds and report mean±SD across trials. Compilation uses GCC13 and Clang17 with comparable C11 flags (`-std=c11 -O2` with strict warnings); any disagreement counts as failure. *pass@1* is the primary outcome in the main text. We ensure that the test prompts used in evaluation do not include any test case information, such as example inputs or expected outputs, to prevent any potential leakage of test-specific details into the decoding process. Specifically, the prompts contain only the problem statement and I/O contract, ensuring fairness in the evaluation across all methods.

4.1 Main Results

Table 1: Performance of base models and *CodeRule-RL* variants on **pass@1** (primary). Across 1.3B–7B backbones, *CodeRule-RL* increases pass@1.

Model	pass@1 (%)
AZR-Coder-3b (Zhao et al., 2025)	15.74
NextCoder-7B (Aggarwal et al., 2025)	36.60
Seed-Coder-8B (ByteDance Seed et al., 2025)	37.45
Deepseek-Coder-1.3B (Guo et al., 2024)	2.13
Deepseek-Coder-1.3B w / *CodeRule-RL*	6.00 (**+3.87**)
Deepseek-Coder-6.7B (Guo et al., 2024)	18.72
Deepseek-Coder-6.7B w / *CodeRule-RL*	28.09 (**+9.37**)
Qwen2.5-Coder-1.5B (Hui et al., 2024)	2.55
Qwen2.5-Coder-1.5B w / *CodeRule-RL*	11.49 (**+8.94**)
Qwen2.5-Coder-3B (Hui et al., 2024)	20.43
Qwen2.5-Coder-3B w / *CodeRule-RL*	22.13 (**+1.70**)
Qwen2.5-Coder-7B (Hui et al., 2024)	21.13
Qwen2.5-Coder-7B w / *CodeRule-RL*	**39.57** (**+18.44**)

Effectiveness of *CodeRule-RL*. Across all backbones (Table 1), *CodeRule-RL* yields consistent absolute gains in **pass@1**: Qwen2.5–Coder–7B 21.13 $\rightarrow$ 39.57 (+18.44; +87.3%), Qwen2.5–Coder–3B 20.43 $\rightarrow$ 22.13 (+1.70; +8.3%), Qwen2.5–Coder–1.5B 2.55 $\rightarrow$ 11.49 (+8.94; +350.6%), DeepSeek–Coder–6.7B 18.72 $\rightarrow$ 28.09 (+9.37; +50.1%), DeepSeek–Coder–1.3B 2.13 $\rightarrow$ 6.00 (+3.87; +181.7%). Gains are largest in absolute terms on mid-sized models (7B/6.7B) and largest in relative terms on smaller models (1.5B/1.3B), suggesting that *per-rule reward shaping with a frequency-aware schedule* is especially helpful under limited capacity yet remains effective for larger backbones. We observe no regressions on any backbone. Improvements hold from 1.3B/1.5B to 7B parameters and across two code-model families (Qwen, DeepSeek), indicating that *per-rule rewards with frequency-aware scheduling* generalize beyond a single backbone and decoding setting. The policy reduces brittle patterns that commonly lead to compile or run failures under stricter warnings or alternative toolchains, which benefits pass@1.

Comparison with post-trained SOTA code LLMs. We compare against size-matched baselines using a unified evaluation protocol that employs greedy decoding (T=0, k=1), unified prompts, identical context limits, unified stop sequences, five fixed seeds, and the same GCC/Clang toolchain; unless noted otherwise, all re-evaluated baselines follow this protocol, with details in Appendix D. At the ∼7B tier, *CodeRule-RL* on Qwen2.5–Coder–7B attains **39.57**, surpassing NextCoder–7B (Aggarwal et al., 2025) at 36.60 (+2.97%) and Seed–Coder–8B at 37.45 (+2.12%). At the ∼3B tier, *CodeRule-RL* on Qwen2.5–Coder–3B reaches **22.13**, exceeding AZR–Coder–3B (Zhao et al., 2025) at 15.74 (+6.39). These size-matched results show that *rule-*

wise reward shaping during RL translates into higher one-shot functional success than strong post-trained baselines of similar size. Our claims of superiority are restricted to models re-evaluated in our unified harness.

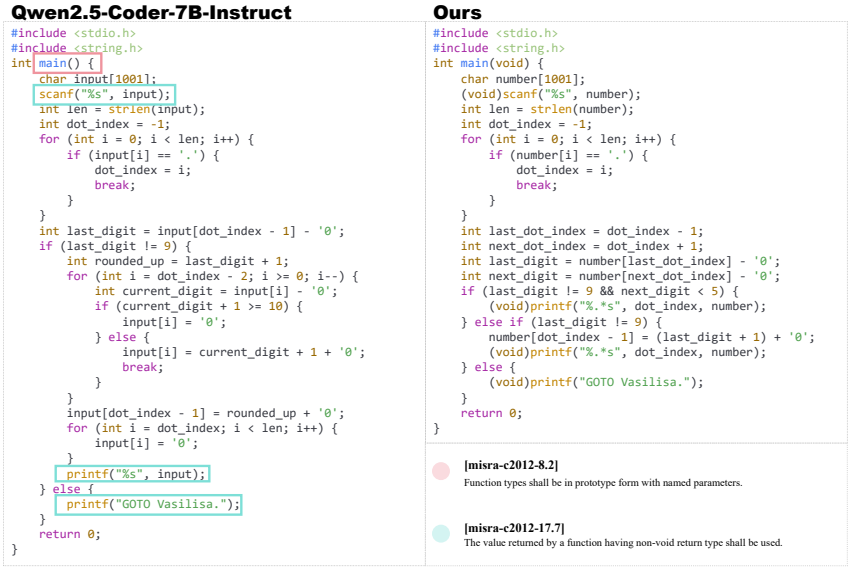

Figure 3: An off-the-shelf output versus the *CodeRule-RL* output on a CodeContests+ task.

Qualitative example. Figure 3 contrasts a typical base decode with the *CodeRule-RL* output on a CodeContests+ task. The base decode compiles and passes tests under the default harness but uses the nonprototype form of `main` and ignores library return values. After training, the policy produces `int main(void)` and acknowledges I/O returns, while preserving problem logic and I/O behavior. In aggregate, such targeted, *per-rule* adjustments correlate with higher **pass@1** (Table 1); we report compliance only as a secondary observation.

4.2 Ablation Study

Comparison with unit test–based RL (pass@1). To isolate the effect of the reward signal, we evaluate *CodeRule-RL* and CURE (Wang et al., 2025a) under a prompt-parity protocol on `Qwen2.5-Coder-3B` using the same split, decoding settings, and context limits. *CodeRule-RL* attains the highest **pass@1** (22.13% vs. 19.15% for CURE; base 5.11%; Table 2). In the *No-Test* setting, where all test-specific text is removed from the prompts, *CodeRule-RL* still reaches 20.85%, showing that gains do not depend on prompt-side exposure to tests. Beyond accuracy, *CodeRule-RL* trains without executing unit tests: rewards are dense, deterministic, and per rule, which removes harness execution overhead and flakiness, improves credit assignment, and simplifies scaling compared with execution-driven RL. These properties make the source of improvement explicit and confer practical advantages in compute cost and training stability.

Training efficiency and design factors. *CodeRule-RL* trains in **1.60** hours versus **21.56** hours for CURE ($\sim$13$\times$ faster) and has lower average reward latency per program (**0.69** s vs. 6.36 s; $\sim$9$\times$ lower). Rewards depend solely on *per-rule* checks from the evaluator introduced in Sec. 3.2 and are computed consistently in batched, cacheable form without executing unit tests during training. A simple frequency-aware schedule inside the reward adjusts only rule weights over time; data, prompts, and the optimizer remain fixed.

Curriculum vs. All Rules. As shown in Figure 4, a frequency-aware schedule over *per-rule* rewards raises **pass@1** more quickly and to a higher plateau than optimizing all rules uniformly: pass@1 approaches $\sim$ 46% for *CodeRule-RL* versus $\sim$ 45% for the all-rules baseline. Auxiliary traces follow the same trend, with the normalized reward rising to $\sim$ 0.90 for *CodeRule-RL* and saturating near $\sim$ 0.81 for the baseline, while the KL trace remains small and stable ($<$ 0.006), indicating controlled updates. Since the optimizer, data, and decoding settings are identical across conditions, the

Table 2: Effect of executing unit tests during training on `Qwen2.5-Coder-3B`. The primary metric is **pass@1**. We compare the base model, CURE (execution-based RL), and *CodeRule-RL*. We also report wall-clock training time (hours) and average reward computation latency (s/sample).

Model	pass@1 (%)	Training time (h)	Latency (s/sample)
Qwen2.5-Coder-3B	5.11	–	–
CURE (Wang et al., 2025a)	19.15	21.56	6.36
CodeRule-RL w/o unit-test prompt	20.85	–	–
CodeRule-RL	**22.13**	**1.60**	**0.69**

improvement is attributable to the reward schedule itself: we adjust only rule weights, emphasizing the most frequently violated rules and gradually relaxing them as violation rates decline.

Rule-wise guidance and pass@1: observational evidence. On `Qwen2.5-Coder-3B` (Hui et al., 2024), *CodeRule-RL* improves **pass@1** even *without* unit-test prompts (20.85% vs. 5.11% base), with a further but modest rise to 22.13% when tests are added. These gains are *consistent with* a mechanism in which coding-standard guidance defines *per-rule* reward shaping and a simple frequency-aware schedule concentrates weight on prevalent violations and then relaxes as they decline. Feedback is consistent and dense during RL; tests remain outside the loop; and only rule weights change while data, prompts, and the optimizer are fixed. This design reduces interference across heterogeneous rules and aligns training pressure with common failure modes, which matches the trend in our curriculum-versus-all-rules comparison (Figure 4) and the overall improvements in Table 1. We do not make a causal claim beyond these associations.

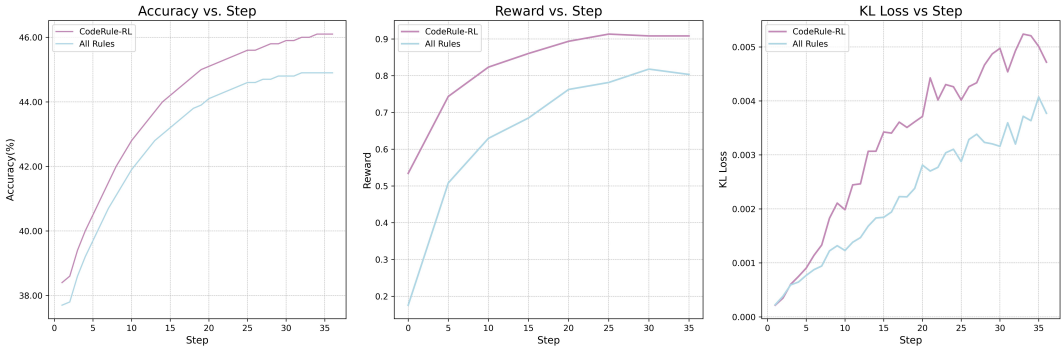

Figure 4: Training dynamics on `Qwen2.5-Coder-7B`: pass@1 (left), compliance reward (middle), and KL loss (right) over training steps for *CodeRule-RL* (curriculum) and the All-Rules baseline (no curriculum). Higher is better for pass@1 and reward; lower is better for KL loss.

5 Conclusion

Prior work has largely overlooked coding-standard guidance as auxiliary signals for training. We use this guidance only to shape per-rule rewards with the sole objective of improving **pass@1**. We introduce ***CodeRule-RL***, a standard agnostic RL framework that keeps *pass@1* as the only optimization target. Machine-checkable rule checks define per-rule reward shaping together with a simple frequency aware schedule implemented inside the reward. The schedule emphasizes the most frequently violated rules first and then gradually reduces their weights as violation rates decline. The optimizer, data, and prompts remain fixed; only reward weights are adjusted. Unit-test specifications may appear in prompts to express requirements, but tests are not executed during training, and we do not construct preference pairs or counterfactual negatives. On a frozen subset of CodeContests+ (C), *CodeRule-RL* attains higher **pass@1** with substantially lower training time than RL that executes tests during training, because rewards are computed from per-rule checks without running tests in the loop.

REFERENCES

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K Arora, Yu Bai, Bowen Baker, Haiming Bao, et al. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*, 2025.

Tushar Aggarwal, Swayam Singh, Abhijeet Awasthi, Aditya Kanade, and Nagarajan Natarajan. Nextcoder: Robust adaptation of code lms to diverse code edits. In *International Conference on Machine Learning*, 2025. URL https://www.microsoft.com/en-us/research/publication/nextcoder-robust-adaptation-of-code-lms-to-diverse-code-edits/.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Roberto Bagnara, Abramo Bagnara, and Patricia M Hill. The misra c coding standard and its role in the development and analysis of safety-and security-critical embedded software. In *International Static Analysis Symposium*, pp. 5–23. Springer, 2018.

Roberto Bagnara, Abramo Bagnara, and Patricia M Hill. A rationale-based classification of misra c guidelines. *arXiv preprint arXiv:2112.12823*, 2021.

Shraddha Govind Barke. *Neuro-Symbolic Program Synthesis for Data-Efficient Learning*. PhD thesis, University of California, San Diego, 2024.

ByteDance Seed, Yuyu Zhang, Jing Su, Yifan Sun, Chenguang Xi, Xia Xiao, Shen Zheng, Anxiang Zhang, Kaibo Liu, Daoguang Zan, Tao Sun, Jinhua Zhu, Shulin Xin, Dong Huang, Yetao Bai, Lixin Dong, Chao Li, Jianchong Chen, Hanzhi Zhou, Yifan Huang, Guanghan Ning, Xierui Song, Jiaze Chen, Siyao Liu, Kai Shen, Liang Xiang, and Yonghui Wu. Seed-Coder: Let the code model curate data for itself, 2025. URL https://arxiv.org/abs/2506.03524.

Swarat Chaudhuri, Kevin Ellis, Oleksandr Polozov, Rishabh Singh, Armando Solar-Lezama, and Yisong Yue. Neurosymbolic programming. *Foundations and Trends in Programming Languages*, 7(3):158–243, 2021. doi: 10.1561/2500000049. URL https://doi.org/10.1561/2500000049.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2018.

Shihan Dou, Yan Liu, Haoxiang Jia, Limao Xiong, Enyu Zhou, Wei Shen, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, et al. Stepcoder: Improve code generation with reinforcement learning from compiler feedback. *arXiv preprint arXiv:2402.01391*, 2024.

Yunlong Feng, Yang Xu, Xiao Xu, Binyuan Hui, and Junyang Lin. Towards better correctness and efficiency in code generation. *arXiv preprint arXiv:2508.20124*, 2025.

Debargha Ganguly, Srinivasan Iyengar, Vipin Chaudhary, and Shivkumar Kalyanaraman. Proof of thought: Neurosymbolic program synthesis allows robust and interpretable reasoning. *arXiv preprint arXiv:2409.17270*, 2024.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024. URL https://arxiv.org/abs/2401.14196.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Jingxuan He and Martin Vechev. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1865–1879, 2023.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

Manvi Jha, Jiaxin Wan, and Deming Chen. Proof2silicon: Prompt repair for verified code and hardware generation via reinforcement learning. *arXiv preprint arXiv:2509.06239*, 2025.

Harrison Lee, Samrat Phatale, Hassan Mansoor, Kellie Ren Lu, Thomas Mesnard, Johan Ferret, Colton Bishop, Ethan Hall, Victor Carbune, and Abhinav Rastogi. RLAIF: Scaling reinforcement learning from human feedback with AI feedback, 2024. URL https://openreview.net/forum?id=AAxIs3D2ZZ.

Yixuan Li, Julian Parsert, and Elizabeth Polgreen. Guiding enumerative program synthesis with large language models. In *International Conference on Computer Aided Verification*, pp. 280–301. Springer, 2024.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

Jiate Liu, Yiqin Zhu, Kaiwen Xiao, Qiang Fu, Xiao Han, Wei Yang, and Deheng Ye. Rltf: Reinforcement learning from unit test feedback. *arXiv preprint arXiv:2307.04349*, 2023a.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36:21558–21572, 2023b.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*, 2024.

Ehsan Mashhadi, Shaiful Chowdhury, Somayeh Modaberi, Hadi Hemmati, and Gias Uddin. An empirical study on bug severity estimation using source code metrics and static analysis. *Journal of Systems and Software*, 217:112179, 2024.

Tom Mens and Tom Tourwe. A survey of software refactoring. *Software Engineering, IEEE Transactions on*, 30:126 – 139, 03 2004. doi: 10.1109/TSE.2004.1265817.

Niels Mündler, Jingxuan He, Hao Wang, Koushik Sen, Dawn Song, and Martin Vechev. Type-constrained code generation with language models. *Proceedings of the ACM on Programming Languages*, 9(PLDI):601–626, 2025.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Bo Shen, Jiaxin Zhang, Taihong Chen, Daoguang Zan, Bing Geng, An Fu, Muhan Zeng, Ailun Yu, Jichuan Ji, Jingyang Zhao, et al. Pangu-coder2: Boosting large language models for code with ranking feedback. *arXiv preprint arXiv:2307.14936*, 2023.

Yinjie Wang, Ling Yang, Ye Tian, Ke Shen, and Mengdi Wang. Co-evolving llm coder and unit tester via reinforcement learning. *arXiv preprint arXiv:2506.03136*, 2025a.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *EMNLP*, 2021.

Zihan Wang, Siyao Liu, Yang Sun, Hongyan Li, and Kai Shen. Codecontests+: High-quality test case generation for competitive programming, 2025b. URL https://arxiv.org/abs/2506.05817.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. Wizardlm: Empowering large pre-trained language models to follow complex instructions. In *The Twelfth International Conference on Learning Representations*, 2024.

Chuanhao Yan, Fengdi Che, Xuhan Huang, Xu Xu, Xin Li, Yizhi Li, Xingwei Qu, Jingzhe Shi, Zhuangzhuang He, Chenghua Lin, Yaodong Yang, Binhang Yuan, Hang Zhao, Yu Qiao, Bowen Zhou, and Jie Fu. Re:form – reducing human priors in scalable formal software verification with rl in llms: A preliminary study on dafny, 2025. URL https://arxiv.org/abs/2507.16331.

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

Andrew Zhao, Yiran Wu, Yang Yue, Tong Wu, Quentin Xu, Yang Yue, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. Absolute zero: Reinforced self-play reasoning with zero data, 2025. URL https://arxiv.org/abs/2505.03335.

Terry Yue Zhuo, Vu Minh Chien, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen GONG, James Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, David Lo, Binyuan Hui, Niklas Muennighoff, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro Von Werra. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=YrycTjllL0.

A THE USE OF LARGE LANGUAGE MODELS (LLMS)

In accordance with ICLR 2026 policies on LLM usage, we disclose that AI assistance was used only during manuscript preparation for surface-level editing. Specifically, we used ChatGPT, DeepSeek, and Grammarly to correct grammatical errors and refine wording. All ideas, claims, experiment designs, analyses, and conclusions are authored and verified by the human authors. We reviewed and edited all AI suggestions before inclusion. No confidential or under-review material, proprietary data, or private code was provided to any AI system. The authors remain fully responsible for the final content of the paper.

B REPRODUCIBILITY STATEMENT

All implementation and evaluation details needed for replication are specified in the paper. We document the exact dataset manifest and fixed splits, the construction of prompts and the prompt-parity *No-Test* protocol, the reward computation pipeline with analyzer versions and flags, compiler toolchains and build options for pass@1, decoding settings evaluated separately (greedy and nucleus), the definition of success and failure, and the full set of training hyperparameters including curriculum thresholds and reward clipping. We report five independent runs per setting with mean±SD, provide hardware configuration, wall-clock training time, and average reward latency, and describe known sources of non-determinism and the controls we apply. Each table and figure references the scripts and logged fields from which it is derived so results can be regenerated from the documented procedures.

C IMPACT OF CODING STANDARDS ON MODEL FUNCTIONALITY

Figure 5 illustrates the positive correlation between the adoption of coding standards and model performance. As training progresses, both the *CodeRule-RL* and All Rules conditions show significant improvements in functionality, with a reduction in program errors. This pattern consistently appears across different experimental setups, confirming the effectiveness of coding standards in enhancing both program quality and model performance.

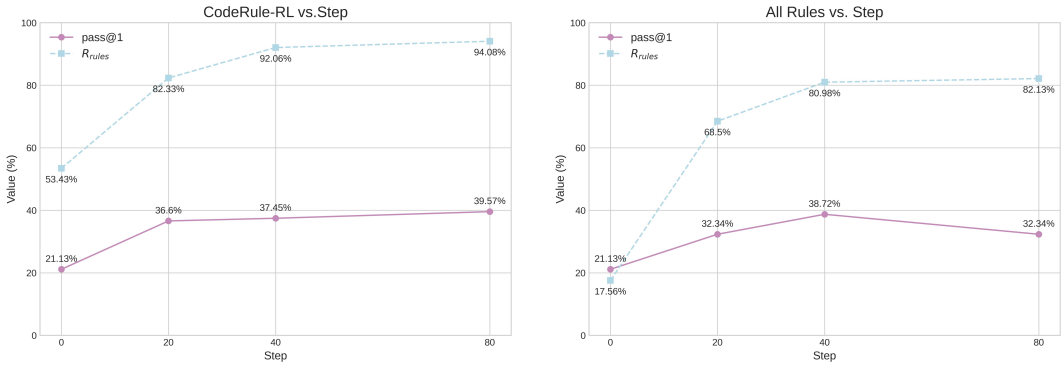

Figure 5: Correlation between functionality and training rewards. Experiments were conducted on the `Qwen2.5-Coder-7B` model. The left plot shows *CodeRule-RL*, and the right plot shows all rules.

D RE-EVALUATION DETAILS FOR BASELINES

All re-evaluated baselines use a unified protocol: greedy decoding (T=0, k=1) with no per-model tuning; input/output context limits identical to ours; a unified set of stop sequences (listed in the appendix); five fixed random seeds across all runs; and the same GPU class with the same compilation toolchain (GCC 13 / Clang 17).

Algorithm 1 *CodeRule-RL*

Input:
1) Initial policy π_θ, reference policy π_{ref}.
2) Number of iterations M.
3) Static analyzer Ψ, rule set $\mathcal{R}$.
4) Base importances $\{\bar{w}_r\}$, coverage mask $\mathbf{m}$, shaping functions $\{\psi_r\}$, caps $\{N_r\}$, gains $\{k_r\}$, EMA rate λ, mastery threshold τ, window W, warmup T_{warm}, cool-down T_{cool}, hysteresis h, gate threshold ϵ_g.
5) GRPO group size N, PPO clip ϵ, KL weight β.
Initialize: $\bar{\mathbf{v}}(0) \leftarrow \mathbf{0}$; $K(0) \leftarrow 1$; set $\alpha_r(0) \leftarrow 0$; clear $\{t_r^{\text{on}}, t_r^{\text{master}}\}$.
for $t = 1$ to M **or** not converged **do**
  Sample a batch of prompts $\{q_j\}_{j=1}^{B}$; for each q_j, sample N candidates $C_{j,1:N} \sim \pi_\theta(\cdot \mid q_j)$
  Run Ψ to obtain the rule-indexed violation vector $\mathbf{v}(C_{j,k})$ for all candidates
  if analysis fails for $C_{j,k}$ **then**
    $R(C_{j,k}, t) \leftarrow R_{\text{fail}}$ ▷ hard penalty for analyzer failure
  end if
  Update curriculum state:
  $\bar{\mathbf{v}}(t) \leftarrow (1-\lambda)\bar{\mathbf{v}}(t-1) + \lambda \cdot \mathbb{E}_{j,k}[\mathbf{v}(C_{j,k})]$ ▷ EMA of per-rule violation frequencies
  Rank rules by $\bar{\mathbf{v}}(t)$; let $\mathcal{S}(t) = \text{Top-}K(t)$ ▷ active set of $K(t)$ most violated rules
  if $\forall r \in \mathcal{S}(t)$ have been under τ for W steps **then**
    $K(t+1) \leftarrow \min\{K(t)+1, K_{\max}\}$; set $t_r^{\text{on}} \leftarrow t$ for newly activated rules
  else
    $K(t+1) \leftarrow K(t)$
  end if
  Apply hysteresis: rules previously deactivated may reenter only if $\bar{v}_r(t) \geq \tau + h$
  Compute rewards:
  for each $C_{j,k}$ **do**
    for each $r \in \mathcal{R}$ **do**
      $\hat{v}_r = \psi_r\big(\min\{v_r(C_{j,k}),\, N_r\}\big)$ ▷ cap & shape the raw violation count
      $P_r(\hat{v}_r) = 1 - \exp(-k_r \hat{v}_r)$ ▷ per-rule penalty in $[0, 1)$
      Compute schedule $\alpha_r(t)$ with warmup (T_{warm}), anneal/cool-down (T_{cool})
      Gate $g_r(t, \mathbf{v}) = \mathbb{1}\{r \in \mathcal{S}(t)\} \cdot \mathbb{1}\{v_r(C_{j,k}) \geq \epsilon_g\}$ ▷ active & triggered
      $w_r(t, \mathbf{v}) = \bar{w}_r\, m_r\, \alpha_r(t)\, g_r(t, \mathbf{v})$ ▷ effective per-rule weight
    end for
    $R(C_{j,k}, t) = \text{clip}\Big(1 - \sum_{r \in \mathcal{R}} w_r(t, \mathbf{v})\, P_r(\hat{v}_r),\, -0.5,\, 1.2\Big)$ ▷ static-analysis-only reward
  end for
  Optimize the policy π_θ:
  For each group j, center advantages $A_{j,k} = R(C_{j,k}, t) - \frac{1}{N}\sum_{u=1}^{N} R(C_{j,u}, t)$
  Update θ with PPO-style clipped objective (clip ϵ), KL penalty $\beta\, \text{KL}(\pi_\theta \| \pi_{\text{ref}})$, and a small entropy bonus
end for
Output: Trained generator π_θ.

E VULNERABILITY DETECTION

Traditional code reviews and dynamic testing often fail to cover all edge cases, leaving potential issues undetected. To provide a more comprehensive evaluation of the generated code, we employ **Infer**, a state-of-the-art static analysis tool. Unlike dynamic testing, which executes the code, Infer conducts an in-depth analysis of the source code to identify potential vulnerabilities and runtime errors that are challenging to detect through conventional methods.

For instance, Infer can track the complete lifecycle of variables, enabling it to flag memory management errors such as improper memory allocation and deallocation. Furthermore, it identifies performance bottlenecks like **`EXPENSIVE_LOOP_INVARIANT_CALL`**. This issue occurs when a computationally expensive function (determined through cost analysis to have at least linear complexity) that is loop-invariant is called inside a loop. This inefficient coding pattern can severely degrade performance, especially when the code is executed repeatedly.

To quantify the security and robustness of the generated code, we introduce the **Vulnerability-Free Rate (VFR)** metric. This metric measures the percentage of code samples that pass the Infer static

analysis scan without any detected issues. It is mathematically defined as:

$$\mathrm{VFR} = \frac{1}{|\mathcal{X}_{\mathrm{eval}}|} \sum_{x \in \mathcal{X}_{\mathrm{eval}}} \mathbb{1}\big[F_{BI}(x) = 1\big]. \tag{10}$$

where, F_{BI} is Facebook Infer vulnerability detector. The VFR complements the **pass@1** metric, which measures functional correctness, to form a comprehensive framework for evaluating model performance.

As shown in Table 3, our experiments reveal that the optimization method generally improves the functional correctness (**pass@1**) of the models. However, its impact on code security (**VFR**) varies across different models. For example, after optimization, `Qwen2.5-Coder-1.5B` shows a **+8.94%** improvement in **pass@1** but a **-3.44%** decrease in its VFR. This suggests that while the new code is more functionally correct, it also introduces more potential issues detectable by static analysis. In contrast, larger models like `Qwen2.5-Coder-7B` demonstrate excellent performance on both fronts, achieving a significant **+18.44%** increase in **pass@1** alongside a solid **+2.66%** improvement in VFR. This result indicates that our optimization method, when applied to larger models, can effectively enhance problem-solving capabilities without sacrificing code quality or security.

Table 3: Performance Comparison of Baseline and Optimized Models (Pass@1 and VFR for Vulnerability Detection)

Model	**pass@1** (%)	**VFR**(%)
Absolute_Zero_Reasoner-Coder-3b	15.74	77.87
NextCoder-7B	36.60	87.02
Qwen3-4B-Instruct-2507	55.32	77.02
Qwen3-4B-Instruct-2507 w / *CodeRule-RL*	56.17 (**+0.85**)	80.43 (**+3.41**)
Deepseek-Coder-1.3B	2.13	41.70
Deepseek-Coder-1.3B w / *CodeRule-RL*	6.00 (**+3.87**)	77.02 (**+35.32**)
Deepseek-Coder-6.7B	18.72	89.36
Deepseek-Coder-6.7B w / *CodeRule-RL*	28.09 (**+9.37**)	90.64 (**+1.28**)
Qwen2.5-Coder-1.5B	2.55	91.91
Qwen2.5-Coder-1.5B w / *CodeRule-RL*	11.49 (**+8.94**)	88.47 (**-3.44**)
Qwen2.5-Coder-3B	20.43	83.02
Qwen2.5-Coder-3B w / *CodeRule-RL*	22.13 (**+1.70**)	88.94 (**+5.92**)
Qwen2.5-Coder-7B	21.13	82.02
Qwen2.5-Coder-7B w / *CodeRule-RL*	39.57 (**+18.44**)	84.68 (**+2.66**)

F Data Processing Pipeline

The pipeline in Figure 6 follows Sec. 3.1 and prepares the data used for training. Tasks are mined and decontaminated, then pass through up to three *probe-and-check* iterations per prompt. Probe candidates that fail to compile are discarded. A rule evaluator Φ produces *per rule* signals $\mathbf{s}(C)$ that are used *only* for offline screening and aggregate quality checks; these signals are *not* stored as labels and do *not* become part of the training corpus. D_{ID} and D_{RM} are temporary inspection sets for reporting and analysis, not supervision. The final frozen training set $\mathcal{D}$ contains prompts and minimal metadata. During RL, per rule signals are recomputed online by Φ on the sampled candidate and are used only inside the reward (Secs. 3.3–3.4); data, prompts, and the optimizer remain fixed.

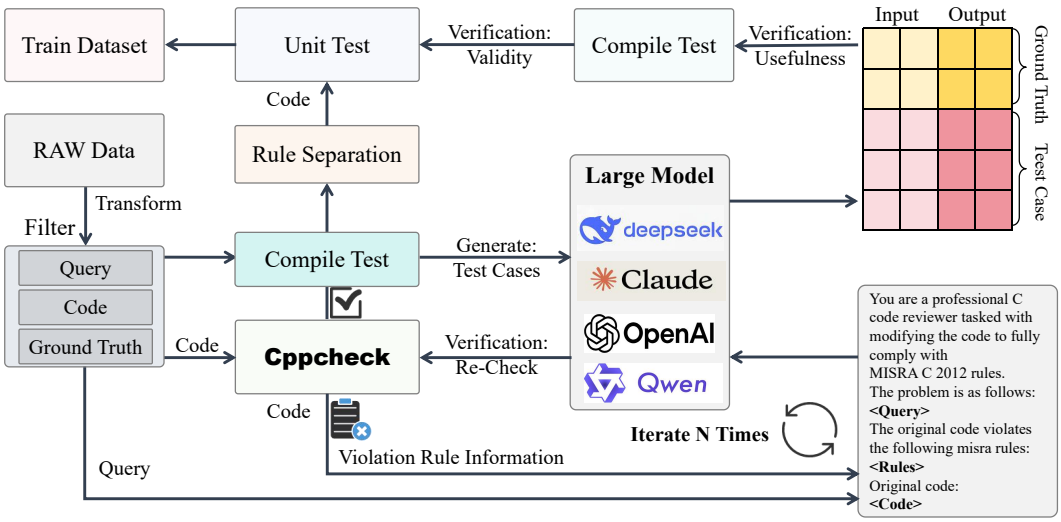

Figure 6: **Data pipeline.** Tasks are mined and decontaminated, followed by up to three probe-and-check iterations for curation. The evaluator Φ produces per rule signals for filtering and diagnostics during curation; these signals are discarded after curation and are not persisted in the dataset. During RL, signals are recomputed online for reward shaping. The pipeline is standard-agnostic: replacing the rule set or checker changes only Φ and its mapping, not the learning algorithm.

G Ablation Study for Compliance

Coding-standard compliance. We assess MISRA C:2012 compliance using `cppcheck` 2.7 with the MISRA addon. With one candidate per prompt (k=1), we report

$$\text{compliance@1} = \frac{1}{|\mathcal{X}_{\text{eval}}|} \sum_{x \in \mathcal{X}_{\text{eval}}} \mathbb{1}\big[\mathcal{C}_{\text{std}}(y_x) = 1\big], \qquad y_x = \text{Decode}_{k=1}(\pi_\theta, x), \tag{11}$$

$$\text{joint@1} = \frac{1}{|\mathcal{X}_{\text{eval}}|} \sum_{x \in \mathcal{X}_{\text{eval}}} \big[\mathbb{1}\big(\Gamma(y_x) = 1 \wedge \mathcal{C}_{\text{std}}(y_x) = 1\big]. \tag{12}$$

Advisory findings are logged but ignored by the metric. Unless otherwise stated, each translation unit is analyzed with `cppcheck` 2.7.

Analysis. To test whether a frequency driven, rule by rule schedule outperforms optimizing all constraints at once, we compare *CodeRule-RL*—a curriculum with a Top-K frontier plus gating and annealing—against an *All Rules* baseline (Figure 7). Under *All Rules*, pass@1 rises modestly ($32.34 \rightarrow 38.72 \rightarrow 34.89$) while compliance@1 remains at 0.43% and joint@1 stays at 0. With *CodeRule-RL*, compliance@1 improves monotonically ($20.00 \rightarrow 42.13 \rightarrow 58.72$), joint@1 increases in step ($8.51 \rightarrow 19.57 \rightarrow 24.68$), and pass@1 remains comparable ($33.19 \rightarrow 40.00 \rightarrow 37.02$). These results support our mechanism: per rule credit assignment and gradient isolation within the active frontier drive compliance and joint success, whereas optimizing all rules simultaneously induces gradient interference and stalls compliance learning.

G.1 HumanEval and MBPP

For our evaluation on general benchmarks, we used **HumanEval**Chen et al. (2021), **HumanEval Plus**, **MBPP**Austin et al. (2021), and **MBPP Plus** to assess Python programming tasks on EvalPlus Liu et al. (2023b). The HumanEval Plus dataset extends the original HumanEval test cases by a factor of 80 to create the HumanEval Plus dataset, while MBPP Plus includes 35 times more test cases than the original MBPP.

Table 4 summarizes the results on HumanEval (and HumanEval Plus) Chen et al. (2021) and MBPP (and MBPP Plus) Austin et al. (2021). After applying our optimization, the 1.5B and 3B models

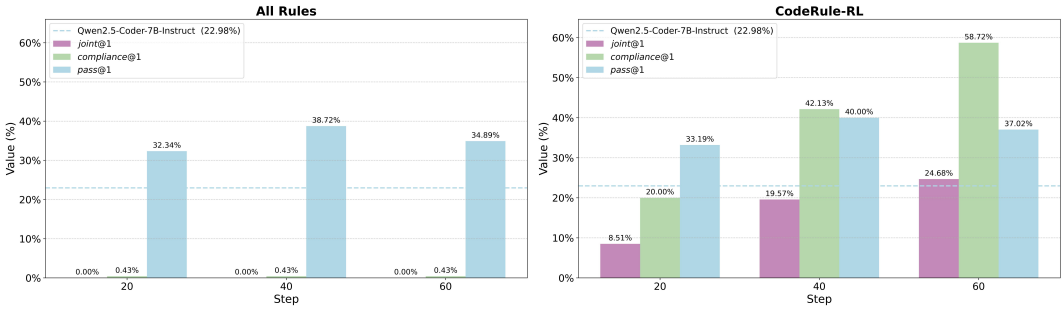

Figure 7: Qwen2.5−Coder−7B under different training regimes and rule granularities. **Single Rule**: optimizes one rule at a time. ***CodeRule-RL***: a frequency driven, phased curriculum that learns multiple rules jointly. **All Rules**: optimizes the full rule set simultaneously.

demonstrate substantial improvements across all metrics. Notably, the 1.5B model shows an over 16% increase in its HE metric, indicating that our method effectively enhances the code-generation capabilities of smaller models. In contrast, the 4B and 7B models exhibit only marginal improvements, likely because these larger models are already approaching the performance ceiling on these benchmarks. Overall, the results suggest that our method incurs little to no degradation in general-purpose coding performance and may even result in slight gains.

Table 4: Performance Comparison of Models on Code Generation Tasks (HumanEval/MBPP, All Metrics in %)

Model	HumanEval		MBPP	
	HE	HE+	MBPP	MBPP+
BASE				
Qwen2.5-Coder-1.5B	59.10	53.70	68.50	58.50
Qwen2.5-Coder-3B	76.20	68.90	70.40	59.30
Qwen3-4B-Instruct-2507	76.80	70.10	80.70	68.30
Qwen2.5-Coder-7B	81.70	75.60	82.30	68.50
Deepseek-coder-1.3B	61.60	57.90	63.80	54.50
Deepseek-coder-6.7B	68.90	63.40	76.20	64.80
Ours				
Qwen2.5-Coder-1.5B	75.60	68.90	70.10	59.50
Qwen2.5-Coder-3B	74.40	67.70	71.20	60.30
Qwen3-4B-Instruct-2507	77.40	72.00	80.70	68.30
Qwen2.5-Coder-7B	81.70	76.20	83.30	68.30
Deepseek-coder-1.3B	60.40	57.30	65.10	56.10
Deepseek-coder-6.7B	69.50	62.80	75.70	64.00

G.2 Annealing warmup Epoch

We use the Qwen2.5-Coder-7B model to compare the impact of different annealing steps on coding standards and functionality. The step size of the annealing process plays a crucial role in learning major violation rules. A larger step size leads to a smaller effective weight for minor rules, thereby prioritizing the learning of coding standards related to major violations. As shown in Table 5, we selected a step size of 30 for our experiments, as it demonstrated the most significant impact on key performance metrics.

Table 5: Influence of Annealing Warmup Epoch

Epoch	join@1 (%)	compliance@1 (%)	pass@1 (%)
0	0.00	0.00	21.13
5	8.09	12.34	35.74
10	7.66	21.70	24.47
20	4.26	6.81	36.60
30	22.26	55.74	39.57
50	4.26	8.51	35.32

H LEARNING SEQUENCE

After the second iteration of the data generation phase (as shown in Figure 6), we selected 4,000 compilable code samples for code style rule violations detection. Based on these samples, we evaluated the model's error rate for different coding rules, with the results shown in Figure 8. The figure illustrates the number of violations for various MISRA C:2012 rules, with `misra-c2012-8.2` having the highest number of violations, reaching 4,473, followed by `misra-c2012-15.6` with 2,240 violations. Most other rules had relatively few violations, reflecting the varying levels of adherence to different rules in coding. Using this violation data, we followed the corresponding sequence for training, detection, and evaluation to fine-tune the model's performance more effectively.

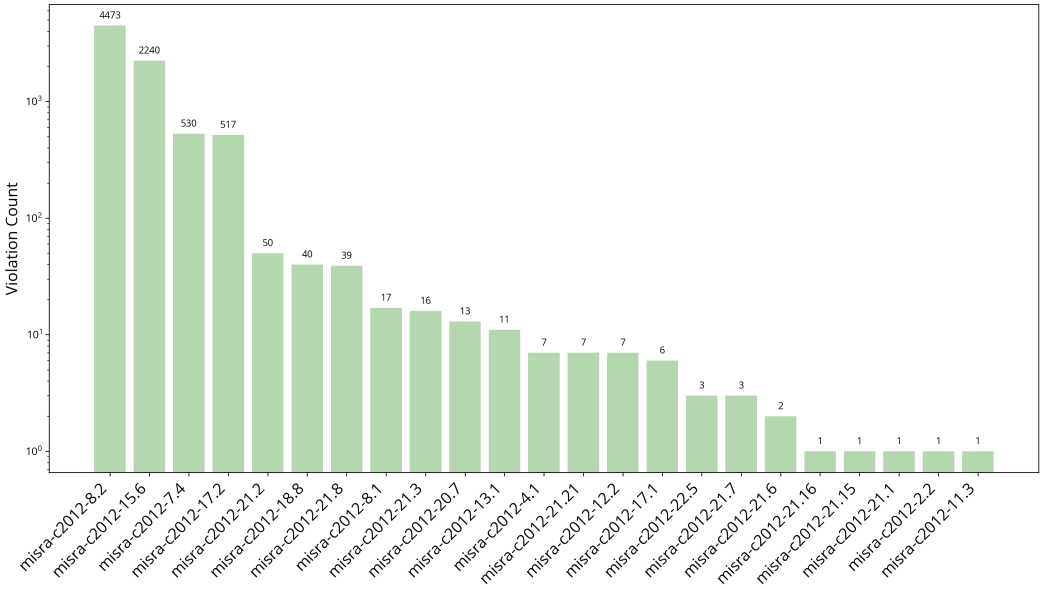

Figure 8: MISRA C:2012 Rule Violation Distribution.

I CODING STANDARDS

As shown in Table 6, we adopt the 178 rules of the MISRA C:2012 standard as the coding guideline (including 18 Mandatory, 122 Required, and 38 Advisory rules) and use the `cppcheck` tool for analysis. Because Advisory rules have minimal impact on coding in software engineering projects, we perform training, detection, and evaluation for the S metric only on the Mandatory and Required categories. Furthermore, since the study uses single-file C data, MISRA C:2012 Rule 21.6 (which prohibits the use of standard library input/output functions in `<stdio.h>` and `<wchar.h>`, such as `printf` and `fgets`) would render `stdio.h` unusable and hinder subsequent file read/write operations needed for evaluation; therefore, this rule was excluded from the process. Ultimately, the

Table 6: MISRA C:2012 rule sets for each category: Mandatory, Required, and Advisory. Counts reflect analyzer outputs, including duplicates. All counts are in percentage.

Category	Rules (IDs)	Count
Mandatory	7.5, 9.1, 12.5, 17.3, 17.4, 17.6, 17.9, 19.1, 21.13, 21.17, 21.18, 21.19, 21.2, 21.22, 22.2, 22.4, 22.5, 22.6	18
Required	1.1, 1.3, 1.4, 1.5, 2.1, 2.2, 3.1, 3.2, 4.1, 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 5.7, 5.8, 6.1, 6.2, 6.3, 7.1, 7.2, 7.3, 7.4, 8.1, 8.2, 8.3, 8.4, 8.5, 8.6, 8.8, 8.12, 8.14, 8.15, 9.2, 9.3, 9.4, 9.5, 10.1, 10.2, 10.3, 10.4, 10.6, 10.7, 10.8, 11.1, 11.2, 11.3, 11.6, 11.7, 11.8, 11.9, 12.2, 13.1, 13.2, 13.5, 13.6, 14.1, 14.2, 14.3, 14.4, 15.2, 15.3, 15.6, 15.7, 16.1, 16.2, 16.3, 16.4, 16.5, 16.6, 16.7, 17.1, 17.2, 17.5, 17.13, 18.1, 18.2, 18.3, 18.6, 18.7, 18.8, 18.9, 20.2, 20.3, 20.4, 20.6, 20.7, 20.8, 20.9, 20.11, 20.12, 20.13, 20.14, 21.1, 21.2, 21.3, 21.4, 21.5, 21.7, 21.8, 21.9, 21.12, 21.14, 21.15, 21.16, 21.21, 21.23, 21.24, 22.1, 22.3, 22.7, 22.8, 22.9, 22.11, 22.15, 22.16, 22.17, 23.2, 23.4, 23.6, 23.8	122
Advisory	1.2, 2.3, 2.4, 2.5, 2.6, 2.7, 4.2, 5.9, 8.7, 8.9, 8.11, 8.13, 8.16, 8.17, 10.5, 11.4, 11.5, 12.1, 12.3, 12.4, 13.3, 13.4, 15.1, 15.4, 15.5, 17.8, 17.11, 17.12, 18.4, 18.5, 19.2, 20.1, 20.5, 21.11, 23.1, 23.3, 23.5, 23.7	38

rules actually included in the evaluation comprise 18 Mandatory and 121 Required rules, for a total of 139 rules.

J Case Examples

As shown in Figs. 10, 3, and 9, we present an illustrative comparison between `Qwen2.5-Coder-7B-Instruct` and our trained model. The three panels respectively show the input prompt, the generated C program, and the `cppcheck` diagnostics. We evaluate on the subset of CodeContests+ tasks, and the evaluation pipeline is fully automated with no manual intervention or post-processing.

Functional correctness. On the illustrated example, our model passes all 34 CodeContests+ unit tests for the task, whereas the `Qwen2.5-Coder-7B-Instruct` baseline passes 19. These counts refer to functional test cases on CodeContests+ and are independent of the static-analysis diagnostics in Figure 9.

Static-analysis diagnostics. Independently of functional testing, we run `cppcheck` and summarize *rule indexed* findings under the analyzer's MISRA C:2012 configuration. Most findings for both models fall under the Required category. Under our accounting (counting Mandatory+Required and excluding Rule 21.6), and without any manual edits, our model reduces the average number of flagged issues by roughly $2\times$ relative to the baseline. Fig. 9 provides the per-rule breakdown.

Qwen2.5-Coder-7B-Instruct

```
test.c:4:9: style: Function types shall be in prototype form with named parameters [misra-c2012-8.2]
int main() {
      ^
test.c:20:35: style: The precedence of operators within expressions should be made explicit [misra-c2012-12.1]
          if (current_digit + 1 >= 10) {
                            ^
test.c:6:10: style: The value returned by a function having non-void return type shall be used [misra-c2012-17.7]
   scanf("%s", input);
       ^
test.c:31:15: style: The value returned by a function having non-void return type shall be used [misra-c2012-17.7]
       printf("%s", input);
            ^
test.c:33:15: style: The value returned by a function having non-void return type shall be used [misra-c2012-17.7]
       printf("GOTO Vasilisa.");
            ^
test.c:1:0: style: The Standard Library input/output functions shall not be used [misra-c2012-21.6]
#include <stdio.h>
^
```

Ours

```
test.c:19:25: style: The precedence of operators within expressions should be made explicit [misra-c2012-12.1]
   if (last_digit != 9 && next_digit < 5) {
                      ^
test.c:1:0: style: The Standard Library input/output functions shall not be used [misra-c2012-21.6]
#include <stdio.h>
^
```

Figure 9: Cppcheck diagnostics for the candidate program.

<|im_start|>**system**
You are Qwen, created by Alibaba Cloud. You are a helpful assistant.
<|im_end|>
<|im_start|>**user**
Please generate a C program that solves the following programming problem:
Mad scientist Mike entertains himself by arranging rows of dominoes. He doesn\'t need dominoes, though: he uses rectangular magnets instead. Each magnet has two poles, positive (a "plus") and negative (a "minus"). If two magnets are put together at a close distance, then the like poles will repel each other and the opposite poles will attract each other.
Mike starts by laying one magnet horizontally on the table. During each following step Mike adds one more magnet horizontally to the right end of the row. Depending on how Mike puts the magnet on the table, it is either attracted to the previous one (forming a group of multiple magnets linked together) or repelled by it (then Mike lays this magnet at some distance to the right from the previous one). We assume that a sole magnet not linked to others forms a group of its own.
<image>
Mike arranged multiple magnets in a row. Determine the number of groups that the magnets formed.
Input
The first line of the input contains an integer n ($1 \le n \le 100000$) — the number of magnets. Then n lines follow. The i-th line ($1 \le i \le n$) contains either characters "01", if Mike put the i-th magnet in the "plus-minus" position, or characters "10", if Mike put the magnet in the "minus-plus" position.
Output
On the single line of the output print the number of groups of magnets.
Examples
Input
6
10
10
10
01
10
10
Output
3
Input
4
01
01
10
10
Output
2

Note
The first testcase corresponds to the figure. The testcase has three groups consisting of three, one and two magnets.
The second testcase has two groups, each consisting of two magnets.
The program should read input from stdin and print output to stdout.
Include necessary headers and write efficient code.
Please provide only the C code in <code> </code>, without any explanations or markdown formatting.
<|im_end|>
<|im_start|>assistant

Figure 10: The example prompt of Codecontents+ (Wang et al., 2025b) for evaluation test cases.