LUA: FUNCTIONAL ALIGNMENT OF LARGE CODE MODELS WITH STREAMING UNIT TESTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large code models (LCMs) have shown strong performance in solving practical software engineering tasks. However, the generated code may not fully align with the user's specification, motivating further alignment steps through reinforcement learning (RL) based finetuning. Recent RL-based finetuning methods for LCMs exploit code-specific properties, particularly the executable property, by using feedback from unit test executions. Yet, existing methods use a limited set of unit tests. In this paper, we investigate how to effectively leverage a large number of unit tests by rethinking curriculum learning as a unit test selection problem. Whilst prior work has applied curriculum learning to order training problems by difficulty, we utilize it to identify informative unit tests from a *large continuously generated pool*. To this end, we propose a lightweight test-difficulty learning layer that estimates the informativeness of individual unit tests, together with an adaptive decoding strategy that selects the tests according to the learned informativeness scores. In particular, we study the scaling behavior of unit-test feedback, the efficiency of test selection, and the need for adaptive selection as the informativeness changes during training. We find that simply executing more tests increases computation cost without necessarily improving performance. In contrast, our adaptive selector effectively tracks informative tests from a *continuously generated pool*, achieving better downstream performance at a lower computational cost. Experiments across diverse LCMs and multiple benchmarks demonstrate that our method effectively leverages a large number of unit tests and consistently improves coding performance over standard reinforcement learning based finetuning.

1 INTRODUCTION

Large language models (LLMs) have demonstrated strong performance in a variety of tasks, including question answering, code generation, and software engineering tasks (Grattafiori et al., 2024; DeepSeek-AI, 2025). Large code models (LCMs), specialized LLMs for code-related tasks, address core challenges in software development, including code summarization (Ahmed et al., 2024; Sun et al., 2025a), code generation (Li et al., 2024; Light et al., 2025; Miserendino et al., 2025), algorithmic problem-solving (Chen et al., 2023; 2024; OpenAI et al., 2025), and security patch generation (Harer et al., 2018; Hu et al., 2025; Kim et al., 2025).

Despite recent performance improvements in LCMs, a discrepancy remains between human expectations and the actual capabilities of the generated code (Tian et al., 2025; Agarwal et al., 2025). Such discrepancies, often referred to as *hallucination in LCMs*, have been documented across various benchmarks designed to capture such phenomena (Chen et al., 2021; Du et al., 2024; Zhu et al., 2025). The *hallucination* problem can be alleviated by different methods based on pre-training (Kalai et al., 2025), post-training (Wei et al., 2024; Gehring et al., 2025), or hallucination control (Jeong et al., 2026). In particular, reinforcement learning (RL) based finetuning methods for mitigating the hallucination problem have received considerable attention for their efficacy (DeepSeek-AI, 2025; OpenAI et al., 2025), although they often incur high computational costs.

In natural language, defining a reliable reward is non-trivial. For example, Ouyang et al. (2022) collect pairwise preference data, whereas Roit et al. (2023) leverage textual entailment (Bowman et al., 2015) as a proxy for correctness and validity. Such approaches therefore often require additional effort in data collection or annotation to construct a reliable reward signal. In contrast, programming

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

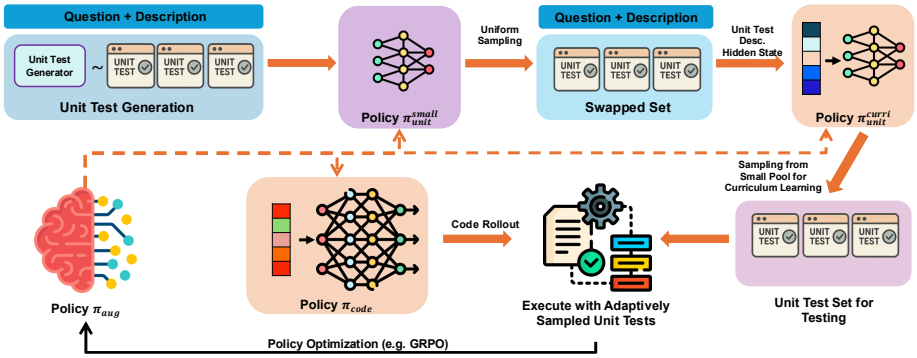


Figure 1: Overview of our proposed policy with decoding and reinforcement learning process. Our method optimizes an augmented policy π_{aug} which integrates π_{code}, $\pi_{\text{unit}}^{\text{small}}$, and $\pi_{\text{unit}}^{\text{curri}}$. Given a prompt and a dynamically expanding pool of unit tests, $\pi_{\text{unit}}^{\text{small}}$ first samples a small subset from the pool. Then, $\pi_{\text{unit}}^{\text{curri}}$ selects the uncertain unit tests to provide informative learning signals. Finally, π_{code} generates code candidates through rollouts during the reinforcement learning process.

languages offer a distinct advantage by leveraging the *executable property* enabled by formally defined syntax and semantics. When code is evaluated on specific inputs, *i.e.,* unit tests, its execution behavior provides a simple and automated measure of functionality, which can be directly used for RL-based finetuning (Team et al., 2025; Jain et al., 2025).

However, the practical applicability of RL-based finetuning of LCMs is hindered by two fundamental limitations: (1) the limited availability of unit tests and (2) the absence of a reliable selection mechanism. In particular, existing datasets typically provide only a limited number of fixed unit tests (Hendrycks et al., 2021), restricting the ability of RL-based finetuning to leverage fine-grained execution-based reward signals. To address this limitation, recent approaches leverage automated test generation techniques, such as fuzzing (Liu et al., 2023b; OpenAI et al., 2025; Team et al., 2025) to augment unit tests for each problem.

Addressing the scarcity of unit tests in this way introduces a new challenge, as the resulting test pools can be computationally expensive to execute during training and may dilute the useful learning signals. This necessitates a reliable mechanism for selecting informative unit tests from a dynamically growing pool. While recent work has explored efficient finetuning strategies for general-purpose LLMs (Yu et al., 2025; Shi et al., 2025; Sun et al., 2025b), these methods do not explicitly consider unit tests, and are not designed to adapt to dynamic environments in which unit tests are continuously generated.

In this paper, to bridge this gap, we propose an adaptive unit test based training framework designed for functional alignment of LCMs. In particular, we *repurpose* curriculum learning (Bengio et al., 2009) as a mechanism for prioritizing large pools of unit tests according to their empirical difficulty. In RL-based finetuning, samples that are neither too easy nor too hard provide the most informative learning signal (Sun et al., 2025b; Yu et al., 2025). Motivated by this observation, we claim that *uncertain* unit tests, those that the generated code may or may not pass, are the most effective for functional alignment. To selectively leverage such tests from a continuously augmented pool, we introduce a test-selection policy for their selection with an adaptive unit test decoding strategy that dynamically identifies informative candidates. Furthermore, we formulate the code generation learning task and unit test decoding task within a unified RL framework, dubbed Large-Scale Unit Test Aligner (LUA). We demonstrate the efficacy of LUA across multiple models and experimental settings, outperforming existing baselines. To ensure reproducibility, all code, datasets, and checkpoints will be released upon publication.

2 Related Work

We review prior work on execution-based LCM finetuning, methods that leverage unit tests, and curriculum learning.

Execution Based Finetuning. The *executable property* of code enables direct observation of program behavior, which can serve as a proxy for correctness. Several works (Le et al., 2022; Shojaee et al., 2023; Liu et al., 2023a) leverage execution signals in RL-based finetuning. Proximal policy optimization (PPO) (Schulman et al., 2017), a policy optimization based on trust region policy optimization (Schulman et al., 2015), has been widely adopted (Shojaee et al., 2023; Jiang et al., 2024; Dou et al., 2024), with execution feedback incorporated as a reward signal. Extending this work, Gehring et al. (2025) apply PPO in a multi-turn setting, while (DeepSeek-AI, 2025; Team et al., 2025; OpenAI et al., 2025) exploit such signals in large-scale models to achieve substantial performance gains. Our method similarly uses execution feedback to finetune LCMs, but addresses the complementary problem of selecting informative unit tests from a large candidate pool.

Unit Tests Leveraged in Finetuning. Finetuning LCMs with execution feedback relies on program execution with specific input test cases. Consequently, using the appropriate input is crucial for obtaining informative execution feedback. Motivated by this, Le et al. (2022); Shojaee et al. (2023); Dou et al. (2024) augment unit tests from the APPS dataset (Hendrycks et al., 2021). Gehring et al. (2025) use the CodeContests dataset provided by Li et al. (2022), which employs a mutation strategy to create test cases. OpenAI et al. (2025) leverages model-generated unit tests during inference but does not disclose how unit tests are used during finetuning, while Team et al. (2025) uses an open-source library to generate a pool of unit tests before finetuning. Existing methods generally rely on a *fixed* unit-test dataset constructed prior to the finetuning stage. In contrast, our method leverages a dynamically expanding pool of generated unit tests and adaptively selects informative tests.

Curriculum Learning. Curriculum learning (Bengio et al., 2009) has been applied to improve general LLM finetuning efficiency by controlling the difficulty of training examples. However, existing approaches primarily operate at the problem level and are not designed to adapt the curriculum at the unit-test level. Shi et al. (2025) rely on predefined difficulty annotations, while Sun et al. (2025b) employ a difficulty-prediction attention-based framework tailored to natural-language and mathematical domains. In the context of LCMs, Hahn et al. (2026); He et al. (2025); Nair et al. (2024) estimate task difficulty from programming language syntax, while other methods (Team et al., 2025; Shi et al., 2025) rely on datasets with fixed unit-test sets and problem-level difficulty labels. In contrast, our approach learns a unit test curriculum without requiring additional difficulty annotations.

3 PROBLEM

We formulate our framework as a token generation problem within a dynamic environment guided by continuously generated unit tests. We present the problem setup in 3.1 and the learning formulation in 3.2.

3.1 PROBLEM SETUP

We consider a practical setting, in which the unit test generator incrementally expands the test suite by generating new unit tests throughout training. This setup is inspired by prior work that generates additional tests on demand (OpenAI et al., 2025; Team et al., 2025; Li et al., 2022). It allows us to evaluate our method when the available test suite continuously evolves throughout training.

3.2 PROBLEM FORMULATION

We view the learning problem as a Markov Decision Process (MDP), defined by the tuple $(\mathcal{S}, \mathcal{A}, P, R, p)$, where $\mathcal{S}$ represents the state space, $\mathcal{A}$ the action space, P the transition kernel, R the reward function, and p the initial state distribution.

Action. The action space $\mathcal{A}$ consists of standard tokens $\mathcal{W}$ and unit test index tokens $\mathcal{U}$. Each unit test index $i \in \mathcal{U}$ is mapped to a specific unit test via `test-fetch`; *i.e.,* `test-fetch`$(i) = (u, v)$ denotes the unit test with index i, described by an input u and an output v. We further denote $\mathcal{A}^*$ as the set of all finite sequences of actions in $\mathcal{A}$.

State. The state space $\mathcal{S}$ represents the sequence of selected actions, defined by $\mathcal{S} := (\mathcal{W} \cup \mathcal{U})^* \subseteq \mathcal{A}^*$. We define the initial state $s_0 \in \mathcal{S}$ as a prompt consisting of a problem and unit test descriptions.

In particular, given a coding problem $\mathbf{x}_{\text{prob}}$ and U unit test descriptions for $\mathbf{x}_{\text{prob}}$, the initial state is formulated as $s_0 \coloneqq (\mathbf{x}_{\text{prob}}, \mathbf{x}_{\text{unit}}^{(1)}, \ldots, \mathbf{x}_{\text{unit}}^{(U)})$, where $\mathbf{x}_{\text{unit}}^{(k)}$ denotes the k-th unit test description.

Transition. The transition kernel $P : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is defined as a deterministic function $P(s, a) \coloneqq [s, a]$, where $[s, a]$ denotes the concatenation of the current state s and the selected action a.

Policy. Building on these notations, we consider a policy $\pi : \mathcal{S} \to \Delta(\mathcal{A})$, where $\Delta(\mathcal{A})$ denotes the set of probability distributions over the action space $\mathcal{A}$. Given the action distribution $\pi(s_t)$, a decoding strategy $d : \Delta(\mathcal{A}) \to \mathcal{A}$ selects the next action as $a_t = d(\pi(s_t))$. We provide the full details of our policy in Section 4.2

Reward. To evaluate the actions generated by the policy, we define a reward function $R(s_h, a_h)$ that primarily measures the functional correctness of the generated output against fetched unit tests. We provide the details of the reward function in 4.3.

Goal. The learning objective is to find an optimal policy π^* that maximizes the expected reward. Formally, we define $\pi^* \in \arg\max_\pi \mathbb{E}\left[R(s_{H_{\text{term}}}, a_{H_{\text{term}}})\right]$, where $H_{\text{term}} \leq H$ denotes the terminal step and H is the maximum horizon. An episode terminates either when the maximum horizon H is reached or an `EOS` token is generated. The expectation is taken over the initial-state distribution p and the sequence of generated actions. While we consider sparse rewards provided at episode termination, this formulation naturally extends to dense reward settings.

4 Aligning LCMs with Large-Scale Unit Tests

We propose an RL-based framework for functional alignment of LCMs that leverages a large-scale pool of unit tests as fine-grained execution feedback. However, exhaustively evaluating a large test pool is computationally expensive and may even degrade performance. To address this bottleneck, we draw inspiration from curriculum learning, and select "uncertain" unit tests to improve learning efficiency.

Figure 1 provides an overview of our framework. We first present our motivation experiments in 4.1, demonstrating the need for our method. 4.2 then formulates the hierarchical policy design, Section 4.3 defines the reward function, and 4.4 details our decoding strategy for curriculum learning. Finally, Section 4.5 and Appendix C present our meta-reinforcement learning algorithm, designed to be compatible with any policy optimization method.

4.1 Motivation: More Unit Tests Are Not Always Better

Table 1: Motivation experiment results on Qwen3-4B with varying number of unit tests used for reward calculation. All results are reported at training step 60.

# Tests (K)	**LCB v5**		**CodeForces**		**HumanEval+**	**Average.**
	pass@1	**pass@8**	**pass@1**	**pass@8**	**pass@1**	
1	26.5	34.9	14.6	27.2	78.4	**36.32**
8	26.0	33.8	13.2	26.6	77.8	35.48
16	26.5	35.3	12.6	26.4	78.0	35.76
Full	25.9	33.5	12.2	23.9	77.3	34.56

We first examine whether using more unit tests for reward computation improves downstream performance. Here, we follow the experimental setup described in Section 5, varying the number of unit tests used to compute the reward defined in (1). In particular, using more unit tests for reward computation can provide broader feedback during RL finetuning. However, our results show that increasing the number of unit tests does not consistently improve downstream performance at the same step. As shown in Table 1, using a single unit test for reward calculation achieves the best average performance of 36.32, while using the full test suite achieves 34.56.

These results indicate that simply scaling the amount of unit-test feedback is insufficient for RL finetuning; this raises the question of *which unit tests to execute for the current policy,* rather than simply scaling the number of tests. We therefore explore whether adapting test selection to the policy

as it evolves during post-training can provide effective signals. This motivates our method, which learns to prioritize informative unit tests from a dynamic unit test pool without executing all tests.

4.2 AUGMENTED POLICY DESIGN

Overview. We formally define our augmented policy π_{aug}, which jointly models unit-test selection and code generation. The augmented policy operates through a three-phase process: (1) candidate test sampling from the generated pool S_t to a small unit test pool by a policy $\pi_{\text{unit}}^{\text{small}}$ (*e.g.,* this policy loads unit tests from slow storage to memory), (2) curriculum test sampling from a small unit test pool for curriculum learning by a policy $\pi_{\text{unit}}^{\text{curri}}$ (*e.g.,* selecting uncertain unit tests for curriculum learning), and (3) code generation using a policy π_{code}.

Formally, we factorize the augmented policy into three components as follows:

$$\pi_{\text{aug}}(\hat{\mathbf{y}}, \mathcal{U}_{\text{curri}}, \mathcal{U}_{\text{small}} \mid s_0) \coloneqq \pi_{\text{unit}}^{\text{small}}(\mathcal{U}_{\text{small}} \mid \mathbf{x}_{\text{prob}}) \cdot \pi_{\text{unit}}^{\text{curri}}(\mathcal{U}_{\text{curri}} \mid s_0, \mathcal{U}_{\text{small}}) \cdot \pi_{\text{code}}(\hat{\mathbf{y}} \mid \mathbf{x}_{\text{prob}}, \mathcal{U}_{\text{curri}})$$

where s_0 denotes the initial problem context associated with $\mathbf{x}_{\text{prob}}$, $\mathcal{U}_{\text{small}}$ is the tractable candidate pool of unit tests filtered from $\mathcal{U}$, $\mathcal{U}_{\text{curri}} \subseteq \mathcal{U}_{\text{small}}$ is the specific unit test set selected for curriculum learning, and $\hat{\mathbf{y}}$ denotes the final sequence of generated code tokens.

We further define action spaces, $\mathcal{A}_{\text{unit}} \coloneqq \mathcal{U}$ and $\mathcal{A}_{\text{code}} \coloneqq \mathcal{W}$, corresponding to unit-test index tokens and code tokens, respectively. For a subset $\mathcal{S} \subseteq \mathcal{A}$, we denote by $\Delta_{\mathcal{S}}(\mathcal{A})$ the simplex over $\mathcal{A}$ whose support is restricted to $\mathcal{S}$.

Phase 1: $\pi_{\text{unit}}^{\text{small}}$. The primary role of the filtering policy $\pi_{\text{unit}}^{\text{small}} : \mathcal{S} \to \Delta_{\mathcal{A}_{\text{unit}}^*}(\mathcal{A})$ is to reduce an intractable unit test pool to a candidate pool $\mathcal{U}_{\text{small}}$. At any given training step t, the full unit test pool may be too large to enumerate, making it impractical to sample from an arbitrary distribution over the entire space. Moreover, we assume that no prior information is available about the informativeness of individual unit tests. Under these constraints, $\pi_{\text{unit}}^{\text{small}}$ uniformly samples a tractable subset.

Phase 2: $\pi_{\text{unit}}^{\text{curri}}$ and $\tilde{\pi}_{\text{unit}}^{\text{curri}}$. The curriculum policy $\pi_{\text{unit}}^{\text{curri}} : \mathcal{S} \to \Delta_{\mathcal{A}_{\text{unit}}^*}(\mathcal{A})$ selects the uncertain unit tests from the reduced pool $\mathcal{U}_{\text{small}}$. We parameterize this selection process using a token-level unit-test policy $\tilde{\pi}_{\text{unit}}^{\text{curri}} : \mathcal{S} \to \Delta_{\mathcal{A}_{\text{unit}}}(\mathcal{A})$.

For each candidate unit test, we pair the problem description with the i-th unit test description $s_{\text{unit}}^{(i)} \coloneqq (\mathbf{x}_{\text{prob}}, \mathbf{x}_{\text{unit}}^{(i)})$. We forward $s_{\text{unit}}^{(i)}$ through the original model and denote the last-token hidden representation by $\phi(s_{\text{unit}}^{(i)}) \in \mathbb{R}^D$, where $\phi : \mathcal{S} \to \mathbb{R}^D$. Using a learnable scoring vector $\mathbf{w}_{\text{unit}} \in \mathbb{R}^D$, we assign each candidate unit test a score $z_i \coloneqq \mathbf{w}_{\text{unit}}^\top \phi(s_{\text{unit}}^{(i)})$. Applying a softmax σ over all candidates in $\mathcal{U}_{\text{small}}$ yields a per-test selection distribution:

$$\tilde{\pi}_{\text{unit}}^{\text{curri}}(a = i \mid s_0, \mathcal{U}_{\text{small}}) \coloneqq \left[\sigma\left(\left\{\mathbf{w}_{\text{unit}}^\top \phi(s_{\text{unit}}^{(j)})\right\}_{j=1}^{|\mathcal{U}_{\text{small}}|}\right)\right]_i$$

where the action a corresponds to selecting the $i-th$ unit test in $\mathcal{U}_{\text{small}}$.

The subset-level curriculum policy $\pi_{\text{unit}}^{\text{curri}}$ is then obtained by drawing a curriculum subset from the per-test distribution $\tilde{\pi}_{\text{unit}}^{\text{curri}}$. In particular,

$$\mathcal{U}_{\text{curri}} \sim \pi_{\text{unit}}^{\text{curri}}(\cdot \mid s_0, \mathcal{U}_{\text{small}})$$

is formed by sampling individual unit tests according to $\tilde{\pi}_{\text{unit}}^{\text{curri}}(\cdot \mid s_0, \mathcal{U}_{\text{small}})$.

Phase 3: π_{code} and $\tilde{\pi}_{\text{code}}$. The coding policy $\pi_{\text{code}} : \mathcal{S} \to \Delta(\mathcal{A}_{\text{code}}^*)$ generates the final code tokens $\hat{\mathbf{y}}$ conditioned on the problem $\mathbf{x}_{\text{prob}}$. We parameterize it through a token-level policy $\tilde{\pi}_{\text{code}} : \mathcal{S} \to \Delta(\mathcal{A}_{\text{code}})$ with a learnable matrix $\mathbf{W}_{\text{code}} \in \mathbb{R}^{|\mathcal{A}_{\text{code}}| \times D}$. At each decoding step h, the state $s_h \coloneqq (\mathbf{x}_{\text{prob}}, \mathcal{U}_{\text{curri}}, \hat{\mathbf{y}}_{<h})$ collects the input context and the tokens generated so far, and the next token distribution is given by

$$\tilde{\pi}_{\text{code}}(a_h \mid s_h) \coloneqq \sigma(\mathbf{W}_{\text{code}} \cdot \phi(s_h)) \,.$$

The sequence-level policy π_{code} is then induced autoregressively.

Although the initial state contains both the problem and unit test descriptions, *i.e.,* $(\mathbf{x}_{\text{prob}}, \mathbf{x}_{\text{unit}}^{(1)}, \ldots, \mathbf{x}_{\text{unit}}^{(U)})$, each policy conditions on only a relevant subset: the coding policy attends to $\mathbf{x}_{\text{prob}}$ alone, while the unit test policy attends to $\mathbf{x}_{\text{prob}}$ paired with a specific unit test.

4.3 Reward Function Design

Our framework dynamically constructs the unit test set for reward evaluation. We explain how this dynamic test set is incorporated into the reward function.

Prior Reward Function. The conventional reward function (Le et al., 2022; Gehring et al., 2025) implicitly evaluates the generated code against a fixed unit test set. We make this dependence explicit by introducing $\mathcal{U}_{\text{fixed}}$ into the formulation:

$$R(s_t, a_t; \mathcal{U}_{\text{fixed}}) \coloneqq \begin{cases} \dfrac{1}{|\mathcal{U}_{\text{fixed}}|} \displaystyle\sum_{(u,v)\in\mathcal{U}_{\text{fixed}}} \mathbb{1}[\hat{\mathbf{y}}(u) = v] & \text{if } a_t = \texttt{EOS}, \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

Our framework. In contrast, we replace the fixed unit test set $\mathcal{U}_{\text{fixed}}$ with a fully dynamic subset $\mathcal{U}_{\text{curri}}$, which changes throughout RL training. At training step t, the reward is defined as:

$$R(s_t, a_t; \mathcal{U}_{\text{curri}}) \coloneqq \begin{cases} \hat{\mathbb{E}}_{(u,v)\sim\mathcal{U}_{\text{curri}}} \left[\mathbb{1}[\hat{\mathbf{y}}(u) = v]\right] & \text{if } a_t = \texttt{EOS} \\ 0 & \text{otherwise} \end{cases}, \tag{2}$$

where (u, v) is the unit test. Crucially, the test cases $\mathcal{U}_{\text{curri}}$ are not fixed but sampled from the curriculum policy $\pi^{\text{curri}}_{\text{unit}}$.

4.4 Adaptive Unit Test Decoding

Under the standard reinforcement learning objective of maximizing the expected return, $\tilde{\pi}^{\text{curri}}_{\text{unit}}$ would tend toward selecting easier unit tests, with higher rewards. Rather than sampling from $\tilde{\pi}^{\text{curri}}_{\text{unit}}$ directly, we define π^{ada} through *adaptive unit test decoding*, which centers the selection on the median of $\tilde{\pi}^{\text{curri}}_{\text{unit}}$ and down-weights candidates with extreme scores.

Tests with intermediate scores lie near the current capability boundary, where the model exhibits less consistent outcomes. To formalize the sampling process, we define the median score as follows:

$$u_{\text{med}} = \textsc{Median}(\{\tilde{\pi}^{\text{curri}}_{\text{unit}}(u \mid s_h) \mid u \in \mathcal{U}_{\text{small}}\}),$$

We transform the original distribution into an adaptive distribution π_{ada} by applying a Gaussian-weighting at the median score.

$$\pi_{\text{ada}}(u \mid s_h) \propto \exp\left(-\frac{\left(\tilde{\pi}^{\text{curri}}_{\text{unit}}(u \mid s_h) - u_{\text{med}}\right)^2}{\sigma}\right),$$

where $\sigma > 0$ controls the width of the weighting kernel.

Finally, we sample a unit test index from $\pi_{\text{ada}}(\cdot \mid s_h)$, called *adaptive unit test decoding* d_{unit}. This decoding strategy is *inherently adaptive*: as the policy learning proceeds, the previously difficult tests become easy, causing their scores to move away from the median as the policy improves. This strategy continuously shifts towards the unit tests that remain near the current capability boundary.

In short, our final decoding strategy is defined as follows:

$$d_{\text{aug}}(\delta) \coloneqq \begin{cases} d_{\text{unit}}(\delta) & \text{if } \delta \in \Delta_{\mathcal{A}_{\text{unit}}}(\mathcal{A}) \\ d_{\text{code}}(\delta) & \text{otherwise} \end{cases}.$$

Here, δ denotes a distribution over actions and d_{code} is a standard decoding method for code.

4.5 Joint Objective for Code Generation and Curriculum Test Selection

Training Objective. The augmented policy is trained to maximize the expected reward. Unlike conventional execution-based finetuning, where the reward is computed from a fixed set of unit tests, our objective depends on a dynamically selected subset of unit tests.

Algorithm. We propose a meta-reinforcement learning algorithm (Algorithm 1) to learn our policy π_{aug} over large-scale unit tests, dubbed Large-Scale Unit Test Aligner (LUA). In particular, our algorithm follows the standard reinforcement learning pipeline along with a few modifications to unit test sampling. The training process employs a standard policy update, denoted as PolicyUpdate in Line 18, which can be instantiated from a standard RL algorithm such as GRPO and PPO.

Algorithm 1 Large-Scale Unit Test Aligner (LUA)

Input: Initial augmented policy π_{aug}, Dataset $\mathcal{D}$, Total epochs E, Filter update interval I_{small}, σ
for $e = 1, \ldots, E$ **do**
  // Phase 1: Periodic Tractable Pool Filtering
  if $e \pmod{I_{\text{small}}} == 0$ **then**
    for each $d \in \mathcal{D}$ **do**
      Sample $\mathcal{U}_{\text{small}} \sim \pi_{\text{unit}}^{\text{small}}(\cdot \mid d)$ under uniform dist.
    end for
  end if
  for context $s_0 \leftarrow (\mathbf{x}_{\text{prob}}, \mathbf{x}_{\text{unit}}^{(1)}, \ldots) \in \mathcal{D}$ **do**
    // Phase 2: Curriculum Selection
    Sample $\mathcal{U}_{\text{curri}} \sim \pi_{\text{ada}}(\cdot \mid s_0, \mathcal{U}_{\text{small}})$
    // Phase 3: Conditional Code Generation Rollout
    for $h = 1, \ldots, H$ **do**
      Sample code token $a_h \sim \pi_{\text{code}}(\cdot \mid s_h)$
      Reward $r_h \leftarrow R(s_h, a_h; \mathcal{U}_{\text{curri}})$ Non-zero only if $a_h =$ EOS
      Update state $s_{h+1} \leftarrow [s_h, a_h]$
    end for
    $\pi_{\text{aug}} \leftarrow \text{POLICYUPDATE}(\pi_{\text{aug}}, r_1, \ldots, r_H)$ Policy optimization algorithm
  end for
end for
return π_{aug}

5 EXPERIMENTS

We empirically evaluate LUA with a learned test-selection policy and an adaptive decoding strategy, across diverse benchmark datasets. We first examine whether simply scaling the number of unit tests improves downstream performance, and then investigate whether informative test selection can achieve stronger performance with fewer test executions. Finally, we analyze why test selection needs to adapt throughout training and how the learned selector behaves as the policy evolves.

5.1 EXPERIMENTAL SETUPS

Training Datasets and LCMs. We used DeepCoder-Preview-Dataset (Luo et al., 2025) as our RL training datasets. We evaluate our method on three models: Qwen3-1.7B, Qwen3-4B (Yang et al., 2025) and Llama-3.2-3B-Instruct (Meta-AI, 2024).

Evaluation Benchmarks. Following the evaluation setup of prior work (Wang et al., 2025), we evaluated our method on the following three benchmarks: HumanEval Plus (Liu et al., 2023b), LiveCodeBench (Jain et al., 2024), and a subset of CodeForces (Mirzayanov, 2015) problems from DeepCoder-Preview-Dataset (Luo et al., 2025). Additional details can be found in Appendix A.

Baseline Methods. To the best of our knowledge, prior work has not explored adaptive curriculum learning at the unit-test level. We therefore compare our method against problem-level curriculum and sample selection methods, which serve as the closest existing baselines to our setting. We additionally compare against RL algorithms designed to improve training efficiency. Additional details on the baseline methods can be found in Appendix B.

5.2 MAIN RESULTS

Overall Performance. Table 2 compares the performance of LUA with existing RL and curriculum learning baselines on Qwen3-4B. Across the evaluated benchmarks, LUA consistently improves code generation performance over the baselines. The gains are observed both on in-distribution (LCB v5, CodeForces) and out-of-distribution (HumanEval+), indicating that the policy provides useful training signals. Furthermore, as demonstrated in Table 3, our model improves performance across different model families and parameter sizes.

Table 2: Main results on Qwen3-4B at training step 60. We compare our method with vanilla GRPO and prior RL-based fine-tuning methods on LiveCodeBench v5 (LCB), CodeForces, and HumanEval+. "+ Ours" denotes applying our method on top of the corresponding baseline.

Method	LCB v5		CodeForces		HumanEval+	Avg.
	pass@1	pass@8	pass@1	pass@8	pass@1	
Qwen3-4B (Base)	23.1	31.1	11.0	24.6	75.4	33.04
Vanilla GRPO	26.0	33.8	13.2	26.6	77.8	35.48
+ **Ours**	27.1	34.8	13.1	26.9	78.1	**36.00** (+0.52)
AdaRFT	25.7	33.9	14.7	28.1	78.6	36.20
+ **Ours**	26.1	34.0	15.4	30.0	79.2	**36.94** (+0.74)
GAIN-RL	24.6	31.8	12.3	24.2	78.9	34.36
+ **Ours**	25.4	33.1	12.8	24.8	77.7	**34.76** (+0.40)

Table 3: Results across models. We compare our method with the original model, vanilla GRPO, and Random-K on LiveCodeBench v5 (LCB), CodeForces, and HumanEval+ using Qwen3-1.7B, Qwen3-4B, and Llama-3.2-3B-Instruct.

Method	LCB v5		CodeForces		HumanEval+	Avg.
	pass@1	pass@8	pass@1	pass@8	pass@1	
Qwen3-1.7B						
Original	13.2	22.2	4.6	12.8	62.6	23.08
Vanilla GRPO	15.7	22.5	5.4	13.4	64.3	24.26
Random-K	15.9	22.2	5.2	12.6	64.4	24.06
Ours	15.5	23.1	5.4	13.5	64.4	**24.38**
Qwen3-4B						
Original	23.1	31.1	11.0	24.6	75.4	33.04
Vanilla GRPO	26.0	33.8	13.2	26.6	77.8	35.48
Random-K	25.7	35.2	13.0	25.4	77.9	35.44
Ours	27.1	34.8	13.1	26.9	78.1	**36.00**
Llama-3.2-3B-Instruct						
Original	6.2	13.8	1.5	4.8	45.5	14.36
Vanilla GRPO	7.6	12.5	1.5	3.8	50.1	15.10
Random-K	7.8	13.1	1.8	4.2	49.5	15.28
Ours	7.9	12.6	2.0	5.2	50.6	**15.66**

Combining Problem- and Test-Level Curriculum. LUA is complementary to existing curriculum learning methods, as it operates at the unit-test level rather than selecting training problems. As demonstrated in Table 2, combining problem-level curriculum methods further improves downstream performance, demonstrating that test-level selection provides additional benefits.

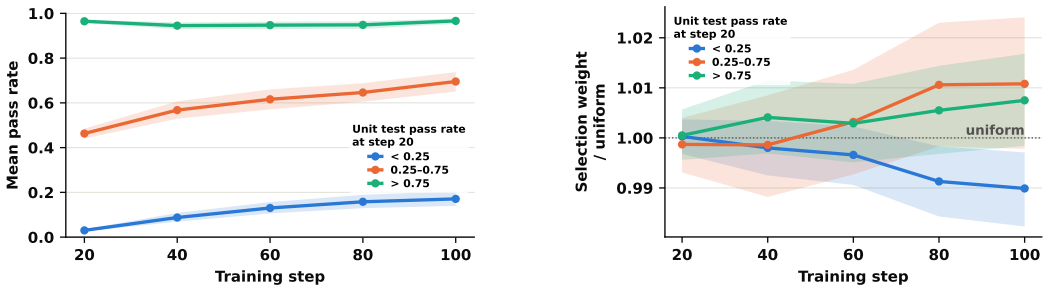
Efficiency of Test Selection. At the same training step, the best fixed single test achieves an average score of 36.32, while combining AdaRFT with LUA achieves 36.94. This demonstrates that the improvement does not come from reducing the number of executed tests, but from selecting informative tests during training.

Effect of Unit Test Augmentation. Table 4 shows a consistent improvement as each component of LUA is introduced. Test augmentation provides an additional gain by training the model with a broader range of correct execution behaviors and execution feedback. Even with uniform sampling, the augmented test pool improves performance, suggesting that the expanded test suite itself provides useful training signals. Combining test augmentation with adaptive selection achieves the best performance, indicating that the two components are complementary. Furthermore, these results are obtained after 40 training steps, yet LUA outperforms the corresponding 60-step results in Table 2.

Table 4: Results on the effect of unit test augmentation at training step 40 with Qwen3-4B. We use the same experimental setup and refresh the candidate set by sampling from the larger unit test pool every five training steps.

	Training		LCB v5		CodeForces		HumanEval+	Avg.
Method	Aug.	Selection	pass@1	pass@8	pass@1	pass@8	pass@1	
Original	–	–	23.1	31.1	11.0	24.6	75.4	33.04
Fixed-Test RL	✗	Fixed	25.4	35.5	14.2	28.4	77.8	36.26
LUA w/o Aug.	✗	Adaptive	26.8	36.3	14.1	28.5	78.4	36.88
LUA w/o Adpt.	✓	Uniform	27.1	35.5	14.9	30.0	77.6	37.02
LUA	✓	Adaptive	26.8	37.1	14.9	29.9	77.7	**37.28**

5.3 How Does LUA Change?



(a) Pass rate (b) Selection weight relative to uniform selection

Figure 2: Candidate tests grouped by their unit test pass rate at Step 20 and tracked throughout training. We randomly sampled 512 problems from the training set and the shaded regions indicate 95% confidence intervals over problems.

We examine how the test selection of LUA changes during training. Figure 2 groups unit tests according to their pass rate at Step 20 and tracks both their pass rates and relative selection weights throughout training. The relative selection weight measures how likely each unit test is to be selected compared with uniform sampling, where a value of 1 corresponds to uniform selection. As the policy improves, the pass rate increases for unit tests that were initially hard or uncertain. Meanwhile, the selection weights of initially hard tests decrease, whereas those of initially uncertain tests increase.

6 Conclusion

In this paper, we presented *Large-Scale Unit Test Aligner* (LUA), a functional alignment framework for large code models that effectively leverages large-scale unit tests. LUA dynamically identifies uncertain unit tests and focuses on them, allowing the model to focus on functional behaviors that remain difficult to pass. Together, our test-selection policy and adaptive decoding strategy enable effective learning from large unit-test pools and can be naturally integrated with problem-level curriculum. Empirical results show that our method scales to large unit tests and improves code generation performance over existing methods.

Limitations. The effectiveness of our algorithm depends on the quality of underlying models and the dataset. If the problem and tests are either too easy or too difficult for the model, the algorithm may not work well. Furthermore, our work is limited to setups where verifiable rewards are available and additional computational resources can be allocated for unit test generations.

AI USE STATEMENT

In this work, we used generative AI tools to assist with writing refinement and code implementation. We have not used generative AI tools for other tasks with required disclosure, and the rest of the required disclosure tasks are not applicable to this work. We have reviewed all AI-assisted work. LLM-generated code was verified and tested for correctness, and AI-assisted writing was reviewed by two authors. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Vibhor Agarwal, Yulong Pei, Salwa Alamir, and Xiaomo Liu. Codemirage: Hallucinations in code generated by large language models, 2025. URL https://arxiv.org/abs/2408.08333.

Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl Barr. Automatic semantic augmentation of language model prompts (for code summarization). In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400702174. doi: 10.1145/3597503.3639183. URL https://doi.org/10.1145/3597503.3639183.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pp. 41–48, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605585161. doi: 10.1145/1553374.1553380. URL https://doi.org/10.1145/1553374.1553380.

Samuel Bowman, Gabor Angeli, Christopher Potts, and Christopher D Manning. A large annotated corpus for learning natural language inference. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pp. 632–642, 2015.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=ktrw68Cmu9c.

Jingchang Chen, Hongxuan Tang, Zheng Chu, Qianglong Chen, Zekun Wang, Ming Liu, and Bing Qin. Divide-and-conquer meets consensus: Unleashing the power of functions in code generation. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=cFqAANINgW.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

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

Shihan Dou, Yan Liu, Haoxiang Jia, Limao Xiong, Enyu Zhou, Wei Shen, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, Zhiheng Xi, Yuhao Zhou, Tao Ji, Rui Zheng, Qi Zhang, Xuanjing Huang, and Tao Gui. Stepcoder: Improve code generation with reinforcement learning from compiler feedback, 2024. URL https://arxiv.org/abs/2402.01391.

Mingzhe Du, Luu Anh Tuan, Bin Ji, Qian Liu, and See-Kiong Ng. Mercury: A code efficiency benchmark for code large language models. *Advances in Neural Information Processing Systems*, 37, 2024.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. RLEF: Grounding code LLMs in execution feedback with reinforcement learning. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=PzSG5nKe1q.

Aaron Grattafiori, Abhimanyu Dubey, and Abhinav Jauhri. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

Joonghyuk Hahn, Aditi, Seung-Yeop Baik, Shinwoo Park, Sang-Ki Ko, and Yo-Sub Han. Encur: Curriculum-based in-context learning with structural encoding for code time complexity prediction. *Expert Systems with Applications*, 296:129094, 2026. ISSN 0957-4174. doi: https://doi.org/10.1016/j.eswa.2025.129094. URL https://www.sciencedirect.com/science/article/pii/S0957417425027113.

Jacob A. Harer, Onur Ozdemir, Tomo Lazovich, Christopher P. Reale, Rebecca L. Russell, Louis Y. Kim, and Peter Chin. Learning to repair software vulnerabilities with generative adversarial networks. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, pp. 7944–7954, Red Hook, NY, USA, 2018. Curran Associates Inc.

Hongkui He, Jiexin Wang, Liuwen Cao, and Yi Cai. Clcosum: Curriculum learning-based code summarization for code language models. In *2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC)*, pp. 1–13. IEEE Computer Society, 2025.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps. *NeurIPS*, 2021.

Yiwei Hu, Zhen Li, Kedie Shu, Shenghua Guan, Deqing Zou, Shouhuai Xu, Bin Yuan, and Hai Jin. Sok: automated vulnerability repair: methods, tools, and assessments. In *Proceedings of the 34th USENIX Conference on Security Symposium*, SEC '25, USA, 2025. USENIX Association. ISBN 978-1-939133-52-6.

Arnav Kumar Jain, Gonzalo Gonzalez-Pumariega, Wayne Chen, Alexander M Rush, Wenting Zhao, and Sanjiban Choudhury. Multi-turn code generation through single-step rewards. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=aJeLhLcsh0.

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.

Jaewoo Jeong, Taesoo Kim, and Sangdon Park. Towards functional correctness of large code models with selective generation, June 2026.

Nan Jiang, Xiaopeng Li, Shiqi Wang, Qiang Zhou, Soneya Binta Hossain, Baishakhi Ray, Varun Kumar, Xiaofei Ma, and Anoop Deoras. Ledex: Training LLMs to better self-debug and explain code. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=d1XrZ4EINV.

Adam Tauman Kalai, Ofir Nachum, Santosh S. Vempala, and Edwin Zhang. Why language models hallucinate, 2025. URL https://arxiv.org/abs/2509.04664.

Taesoo Kim, HyungSeok Han, Soyeon Park, Dae R. Jeong, Dohyeok Kim, Dongkwan Kim, Eunsoo Kim, Jiho Kim, Joshua Wang, Kangsu Kim, Sangwoo Ji, Woosun Song, Hanqing Zhao, Andrew Chin, Gyejin Lee, Kevin Stevens, Mansour Alharthi, Yizhuo Zhai, Cen Zhang, Joonun Jang, Yeongjin Jang, Ammar Askar, Dongju Kim, Fabian Fleischer, Jeongin Cho, Junsik Kim, Kyungjoon Ko, Insu Yun, Sangdon Park, Dowoo Baik, Haein Lee, Hyeon Heo, Minjae Gwon, Minjae Lee, Minwoo Baek, Seunggi Min, Wonyoung Kim, Yonghwi Jin, Younggi Park, Yunjae Choi, Jinho Jung, Gwanhyun Lee, Junyoung Jang, Kyuheon Kim, Yeonghyeon Cha, and Youngjoon Kim. Atlantis: Ai-driven threat localization, analysis, and triage intelligence system, 2025. URL https://arxiv.org/abs/2509.14589. Cross-listed in cs.AI.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Hoi. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (eds.), *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=WaGvb7OzySA.

Jia Li, Ge Li, Xuanming Zhang, Yunfei Zhao, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, and Yongbin Li. Evocodebench: An evolving code generation benchmark with domain-specific evaluations. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2024. URL https://openreview.net/forum?id=kvjbFVHpny.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022. doi: 10.1126/science.abq1158. URL https://www.science.org/doi/abs/10.1126/science.abq1158.

Jonathan Light, Yue Wu, Yiyou Sun, Wenchao Yu, Yanchi Liu, Xujiang Zhao, Ziniu Hu, Haifeng Chen, and Wei Cheng. SFS: Smarter code space search improves LLM inference scaling. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=MCHuGOkExF.

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

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. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023b. URL https://openreview.net/forum?id=1qvx610Cu7.

Michael Luo, Sijun Tan, Roy Huang, Ameen Patel, Alpay Ariyak, Qingyang Wu, Xiaoxiang Shi, Rachel Xin, Colin Cai, Maurice Weber, Ce Zhang, Li Erran Li, Raluca Ada Popa, and Ion Stoica. Deepcoder: A fully open-source 14b coder at o3-mini level, 2025. URL https://pretty-radio-b75.notion.site/DeepCoder-A-Fully-Open-Source-14B-Coder-at-O3-mini-Level-1cf81902c14680b3bee5eb349a512a51. Notion Blog.

Meta-AI. Llama 3.2: Revolutionizing edge ai and vision with open, customizable models, 2024. URL https://ai.meta.com/blog/llama-3-2-connect-2024-vision-edge-mobile-devices/.

Mike Mirzayanov. Open codeforces rating system, 2015. URL https://codeforces.com/blog/entry/20762.

Samuel Miserendino, Michele Wang, Tejal Patwardhan, and Johannes Heidecke. SWE-lancer: Can frontier LLMs earn $1 million from real-world freelance software engineering? In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=xZXhFg43EI.

Marwa Nair, Kamel Yamani, Lynda Lhadj, and Riyadh Baghdadi. Curriculum learning for small code language models. In Xiyan Fu and Eve Fleisig (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 4: Student Research Workshop)*, pp. 390–401, Bangkok, Thailand, August 2024. Association for Computational Linguistics. ISBN 979-8-89176-097-4. doi: 10.18653/v1/2024.acl-srw.44. URL https://aclanthology.org/2024.acl-srw.44/.

OpenAI, :, Ahmed El-Kishky, Alexander Wei, Andre Saraiva, Borys Minaiev, Daniel Selsam, David Dohan, Francis Song, Hunter Lightman, Ignasi Clavera, Jakub Pachocki, Jerry Tworek, Lorenz Kuhn, Lukasz Kaiser, Mark Chen, Max Schwarzer, Mostafa Rohaninejad, Nat McAleese, o3 contributors, Oleg Mürk, Rhythm Garg, Rui Shu, Szymon Sidor, Vineet Kosaraju, and Wenda Zhou. Competitive programming with large reasoning models, 2025. URL https://arxiv.org/abs/2502.06807.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback, 2022.

Paul Roit, Johan Ferret, Lior Shani, Roee Aharoni, Geoffrey Cideron, Robert Dadashi, Matthieu Geist, Sertan Girgin, Leonard Hussenot, Orgad Keller, Nikola Momchev, Sabela Ramos Garea, Piotr Stanczyk, Nino Vieillard, Olivier Bachem, Gal Elidan, Avinatan Hassidim, Olivier Pietquin, and Idan Szpektor. Factually consistent summarization via reinforcement learning with textual entailment feedback. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (eds.), *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 6252–6272, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.344. URL https://aclanthology.org/2023.acl-long.344/.

John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In Francis Bach and David Blei (eds.), *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pp. 1889–1897, Lille, France, 07–09 Jul 2015. PMLR. URL https://proceedings.mlr.press/v37/schulman15.html.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. URL https://arxiv.org/abs/1707.06347.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, pp. 1279–1297, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400711961. doi: 10.1145/3689031.3696075. URL https://doi.org/10.1145/3689031.3696075.

Taiwei Shi, Yiyang Wu, Linxin Song, Tianyi Zhou, and Jieyu Zhao. Efficient reinforcement finetuning via adaptive curriculum learning, 2025. URL https://arxiv.org/abs/2504.05520.

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

Weisong Sun, Yun Miao, Yuekang Li, Hongyu Zhang, Chunrong Fang, Yi Liu, Gelei Deng, Yang Liu, and Zhenyu Chen. Source code summarization in the era of large language models. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, ICSE '25, pp. 1882–1894. IEEE Press, 2025a. ISBN 9798331505691. doi: 10.1109/ICSE55347.2025.00034. URL https://doi.org/10.1109/ICSE55347.2025.00034.

Yifan Sun, Jingyan Shen, Yibin Wang, Tianyu Chen, Zhendong Wang, Mingyuan Zhou, and Huan Zhang. Improving data efficiency for llm reinforcement fine-tuning through difficulty-targeted online data selection and rollout replay. *arXiv preprint arXiv:2506.05316*, 2025b.

Kimi Team, Angang Du, Bofei Gao, Bowei Xing, Changjiu Jiang, Cheng Chen, Cheng Li, Chenjun Xiao, Chenzhuang Du, Chonghua Liao, Chuning Tang, Congcong Wang, Dehao Zhang, Enming Yuan, Enzhe Lu, Fengxiang Tang, Flood Sung, Guangda Wei, Guokun Lai, Haiqing Guo, Han Zhu, Hao Ding, Hao Hu, Hao Yang, Hao Zhang, Haotian Yao, Haotian Zhao, Haoyu Lu, Haoze Li, Haozhen Yu, Hongcheng Gao, Huabin Zheng, Huan Yuan, Jia Chen, Jianhang Guo, Jianlin Su, Jianzhou Wang, Jie Zhao, Jin Zhang, Jingyuan Liu, Junjie Yan, Junyan Wu, Lidong Shi, Ling Ye, Longhui Yu, Mengnan Dong, Neo Zhang, Ningchen Ma, Qiwei Pan, Qucheng Gong, Shaowei Liu, Shengling Ma, Shupeng Wei, Sihan Cao, Siying Huang, Tao Jiang, Weihao Gao, Weimin Xiong, Weiran He, Weixiao Huang, Weixin Xu, Wenhao Wu, Wenyang He, Xianghui Wei, Xianqing Jia, Xingzhe Wu, Xinran Xu, Xinxing Zu, Xinyu Zhou, Xuehai Pan, Y. Charles, Yang Li, Yangyang Hu, Yangyang Liu, Yanru Chen, Yejie Wang, Yibo Liu, Yidao Qin, Yifeng Liu, Ying Yang, Yiping Bao, Yulun Du, Yuxin Wu, Yuzhi Wang, Zaida Zhou, Zhaoji Wang, Zhaowei Li, Zhen Zhu, Zheng Zhang, Zhexu Wang, Zhilin Yang, Zhiqi Huang, Zihao Huang, Ziyao Xu, Zonghan Yang, and Zongyu Lin. Kimi k1.5: Scaling reinforcement learning with llms, 2025. URL https://arxiv.org/abs/2501.12599.

Yuchen Tian, Weixiang Yan, Qian Yang, Xuandong Zhao, Qian Chen, Wen Wang, Ziyang Luo, Lei Ma, and Dawn Song. Codehalu: investigating code hallucinations in llms via execution-based verification. In *Proceedings of the Thirty-Ninth AAAI Conference on Artificial Intelligence and Thirty-Seventh Conference on Innovative Applications of Artificial Intelligence and Fifteenth Symposium on Educational Advances in Artificial Intelligence*, AAAI'25/IAAI'25/EAAI'25. AAAI Press, 2025. ISBN 978-1-57735-897-8. doi: 10.1609/aaai.v39i24.34717. URL https://doi.org/10.1609/aaai.v39i24.34717.

Qinsi Wang, Jinghan Ke, Hancheng Ye, Yueqian Lin, Yuzhe Fu, Jianyi Zhang, Kurt Keutzer, Chenfeng Xu, and Yiran Chen. Angles don't lie: Unlocking training-efficient RL through the model's own signals. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=KGt0F2yjBz.

Yuxiang Wei, Federico Cassano, Jiawei Liu, Yifeng Ding, Naman Jain, Zachary Mueller, Harm de Vries, Leandro Von Werra, Arjun Guha, and LINGMING ZHANG. Selfcodealign: Self-alignment for code generation. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=xXRnUU7xTL.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Mu Qiao, Yonghui Wu, and Mingxuan Wang. Dapo: An open-source llm reinforcement learning system at scale, 2025. URL https://arxiv.org/abs/2503.14476.

Yuxuan Zhu, Antony Kellermann, Dylan Bowman, Philip Li, Akul Gupta, Adarsh Danda, Richard Fang, Conner Jensen, Eric Ihli, Jason Benn, Jet Geronimo, Avi Dhir, Sudhit Rao, Kaicheng Yu, Twm Stone, and Daniel Kang. CVE-bench: A benchmark for AI agents' ability to exploit real-world web application vulnerabilities. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=3pk0p4NGmQ.

A Training Configuration

Training Configuration. The following table demonstrates the configuration used for training. We conducted experiments on four B200 GPUs.

Table 5: Training configuration

Metric	Value
Number of GPUs	4
Training batch size	1024
Maximum prompt length	8192
Maximum response length	4096
Learning rate	1e-6
KL loss coefficient	0.001
Entropy coefficient	0.001
Rollout constant	8
KL coefficient	0.00
Rollout tensor model parallel size	1
PPO max token len per GPU	500,000

Implementation and Hyperparameter Details. We adopt GRPO (DeepSeek-AI, 2025) as our policy optimization algorithm, implemented with VERL framework (Sheng et al., 2025). For all experiments, we used a batch size of 1,024 for all experiments, with a group size of 8 and learning rate of 10^{-6} and σ set to 0.1.

B Baseline

We compare LUA against the following baseline methods.

- **Vanilla GRPO** (DeepSeek-AI, 2025): Vanilla GRPO serves as the standard RL baseline without curriculum learning or selection.
- **AdaRFT** (Shi et al., 2025): AdaRFT adaptively adjusts the sampling of problems using external difficulty labels, providing a problem-level curriculum baseline.
- **GainRL** (Wang et al., 2025): GainRL selects the training problem using the model's internal signal, providing a problem-level curriculum baseline.
- **Random-K**: Random-K randomly selects K unit tests from the available test pool for each problem, serving as a unit-test level random selection baseline.

C Policy Implementation

Each policy is implemented as an auxiliary linear head attached to the last layer of a base LCM. Specifically, π_{code} and $\pi_{\text{unit}}^{\text{small}}$ share a common backbone while each maintains its own lightweight head. This design introduces minimal additional parameters on top of the base model and ensures that all policies operate on a consistent embedding space. Moreover, all policies can be jointly optimized with a single reward signal, eliminating the need for policy-specific reward designs.

D LLM Usage

LLMs were used to assist with code implementation and writing refinement.