ExecVerify: White-Box RL with Verifiable Stepwise Rewards for Code Execution Reasoning

Anonymous ACL submission

Abstract

Code LLMs still struggle with code execution reasoning, especially in smaller models. Existing methods rely on supervised fine-tuning (SFT) with teacher-generated explanations, primarily in two forms: (1) input–output (I/O) prediction chains and (2) natural-language descriptions of execution traces. However, intermediate execution steps cannot be explicitly verified during SFT, so the training objective can be reduced to merely matching teacher explanations. Moreover, training data is typically collected without explicit control over task difficulty. We introduce *ExecVerify*, which goes beyond text imitation by incorporating verifiable white-box rewards derived from execution traces, including next-statement prediction and variable value/type prediction. Our work first builds a dataset with multiple difficulty levels via constraint-based program synthesis. Then, we apply reinforcement learning (RL) to reward correct answers about both intermediate execution steps and final outputs, aligning the training objective with semantic correctness at each execution step. Finally, we adopt a two-stage training pipeline that first enhances execution reasoning and then transfers to code generation. Experiments demonstrate that a 7B model trained with *ExecVerify* achieves performance comparable to 32B models on code reasoning benchmarks and improves pass@1 by up to 5.9% on code generation tasks over strong post-training baselines[1].

1 Introduction

Recent advances in large language models (LLMs) (Hui et al., 2024; Zhu et al., 2024) have achieved strong performance on multiple programming tasks (Jiang et al., 2024; Liu et al., 2023b; Husein et al., 2025). However, these models often struggle to reason about the concrete execution process of programs (Gu et al., 2024b). This limitation hinders semantic understanding and degrades downstream performance on code generation (Gu et al., 2024a) and program repair (Ni et al., 2024; Ye et al., 2022). A key reason is that the training data is predominantly static text (e.g., source code and docstrings) (Luo et al., 2023; Kocetkov et al., 2022).

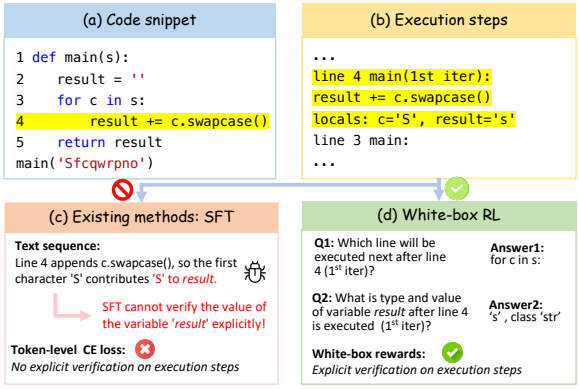

Figure 1: Comparison between SFT and white-box RL. (a) Code snippet. (b) Execution steps extracted from the interpreter, with the relevant parts highlighted in yellow. (c) SFT optimizes the cross-entropy loss over the entire sequence, without explicitly verifying execution details like variable values or control flow. (d) In contrast, white-box RL leverages interpreter-provided execution steps to assign verifiable and step-level rewards.

To bridge this gap, prior work has incorporated execution signals into training, primarily through two approaches: I/O-centric methods (e.g., SEMCODER (Ding et al., 2024a), CODEI/O (Li et al., 2025)), which use execution to validate teacher-generated input–output reasoning chains, and trace-centric methods (e.g., TracePile (Chen et al., 2025), Code Execution as Grounded Supervision (Jung et al., 2025)), which convert execution traces into step-by-step explanations. However, both approaches typically rely on SFT over teacher-written text. Under a token-level cross-entropy

[1]We have released our code, data, and models at https://github.com/justForAnonymous/ExecVerify

objective, intermediate execution steps are not explicitly verified during training. Figure 1 illustrates this limitation of SFT and compares it with our white-box RL approach. As a result, models may overfit to the teacher's textual explanations without truly understanding the execution process. Furthermore, SFT has shown limited generalization ability (Gupta et al., 2025; Wang et al., 2022). In addition, training data is often passively collected or generated without control over difficulty, resulting in many examples that are either trivial or unsolvable, and lacking a structured learning curriculum (see Appendix A.1).

We introduce *ExecVerify*, a framework that enhances execution reasoning by combining Constraint-Based Data Synthesis and White-Box Reinforcement Learning. First, we synthesize programs under explicit structural constraints to construct a curriculum-style dataset with multiple difficulty levels, covering a broad range of commonly used data types and built-in methods. Next, as shown in Figure 1, we convert interpreter traces into verifiable white-box questions that target intermediate control flow, as well as variable types and values. We then apply reinforcement learning (RL) to reward the model for correct predictions on both intermediate steps and final outputs, shifting the objective from text-level imitation to semantic understanding of the execution process. Finally, we adopt a two-stage post-training strategy: the first stage strengthens execution reasoning through white-box rewards, and the second adapts the model to code generation using unit-test feedback, enabling effective transfer from reasoning to generation.

Extensive experiments demonstrate the effectiveness of *ExecVerify*. On execution reasoning benchmarks, a 7B model trained with *ExecVerify* achieves strong results on CRUXEval (Gu et al., 2024b), LiveCodeBench-Exec (Jain et al., 2024), and REval (Chen et al., 2024), and is competitive with much larger models such as Qwen2.5-Coder-32B-Instruct (Hui et al., 2024). Building on this foundation model, when further post-trained for code generation, our model consistently outperforms strong post-training baselines on mainstream benchmarks, including EvalPlus (Liu et al., 2023c), LiveCodeBench (Jain et al., 2024), and BigCodeBench (Zhuo et al., 2024), yielding up to a 5.9% improvement in pass@1.

2 *ExecVerify*

We propose *ExecVerify*, as shown in Figure 2, which improves the LLM's ability in code execution reasoning via **Constraint-Based Data Synthesis (upper part)** and **Two-Stage Post-Training (bottom part)**. *ExecVerify* first synthesizes programs with controlled difficulty under structural constraints. It then applies the Two-Stage Post-Training pipeline. Step one uses verifiable white-box rewards from execution traces for code execution reasoning and step two utilizes unit-test rewards for code generation.

2.1 Constraint-Based Data Synthesis

The goal of Constraint-Based Data Synthesis is twofold: (i) to ensure structural diversity by systematically covering common types, methods, and control-flow patterns; and (ii) to ensure controlled difficulty by generating programs across multiple difficulty levels that remain challenging yet solvable for smaller models. This contrasts with prior methods that collect data without control over structure and difficulty. This component corresponds to the upper part of Figure 2.

2.1.1 Prompt with Constraints

Iterating types and methods. *ExecVerify* begins by iterating over all built-in Python types and their associated methods. For each method, the LLM is explicitly asked to generate a piece of code that must contain the mentioned type and the method.

Generating constraints. To increase complexity, we incrementally apply two types of structural constraints during prompting: **(i) Method-call constraints**, which require the LLM to use nested calls and combine multiple methods within a single function, encouraging rich method interactions; and **(ii) Control-structure constraints**, which enforce the presence of specific nested control-flow patterns, such as `while`, `for`, or `if` statements, to produce non-trivial execution paths (see Appendix B.1 for examples).

From simplicity to complexity. These constraints are introduced in stages: starting with simple code that uses a single method, we then apply method-call constraints, and finally add control structures with increasing nesting depth. This process produces programs that evolve naturally from simple to complex.

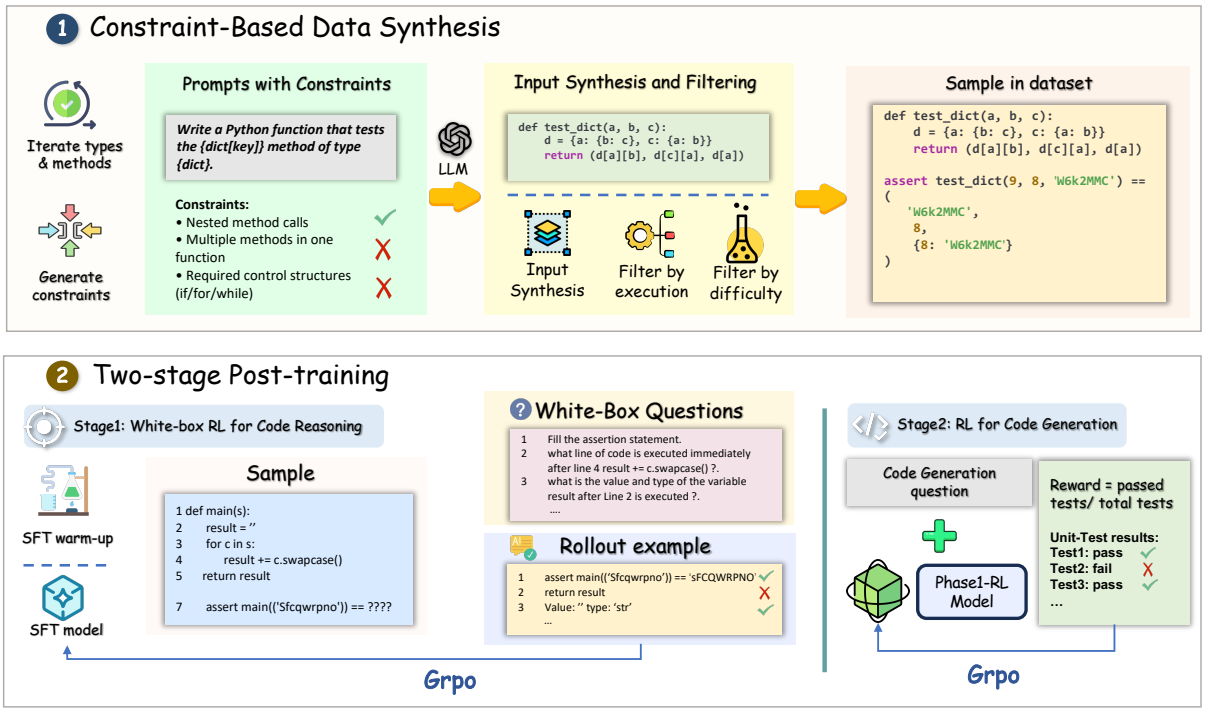

Figure 2: Overview of our approach. Step 1 constructs a constraint-based dataset of executable Python snippets. Step 2 performs two-stage post-training: white-box RL for code reasoning followed by RL for code generation.

2.1.2 Input Synthesis and Data Filtering

Input synthesis. To probe program behavior, we generate diverse inputs for each code snippet. We first prompt an LLM to produce an initial input (as an assertion on the entry-point function), and then apply type-aware mutation following Liu et al. (Liu et al., 2023b) to obtain additional valid inputs. This yields multiple executable inputs per snippet, including both original and mutated variants. In total, we generate **239,992** raw and **239,466** mutated instances before filtering (see Appendix A.4 and Appendix B.2).

Filtering by execution. We execute each synthesized program on all its candidate inputs and discard instances that fail to run successfully or violate basic output constraints (e.g., runtime exceptions, timeouts, or excessively long outputs). As a result, this filtering process leads to **201,537** raw and **191,463** mutated instances.

Filtering by difficulty. *ExecVerify* encourages our model to learn from challenging data points. To filter out trivial samples, we evaluate each remaining instance using Qwen2.5-Coder-7B-Instruct (Hui et al., 2024) under the input–output prediction setting. Specifically, we run the model ten times at temperature 1.0 and count how many predictions pass the test cases. We retain only instances with at most three successful runs (pass count ≤ 3), resulting in **119,358** training examples in total. This yields instances that are non-trivial yet solvable for small models. We report the resulting difficulty and complexity distributions in Appendix A.2–A.4.

Contamination analysis. We also perform an embedding-based contamination analysis against all test sets and find no instances exceeding a conservative similarity threshold (see Appendix A.5).

2.2 Two-stage Post-training

As shown in the bottom part of Figure 2, our training pipeline consists of two stages: Stage I enhances execution reasoning using white-box rewards, while Stage II adapts the model to code generation through unit-test feedback.

2.2.1 Stage I: White-box RL for Code Reasoning

The goal of Stage I is to strengthen the execution reasoning ability by training the model to predict both intermediate execution states and final outputs. This shifts learning from the prior paradigm of imitating teacher explanations to stepwise semantic correctness during execution.

Our work starts with a brief warm-up to inject execution-aware reasoning patterns. Specifically,

we apply supervised fine-tuning (SFT) on input-output prediction reasoning chains generated by a strong teacher model (Team, 2024) and filtered via rejection sampling to ensure correctness. This warm-up provides the model with fundamental execution-relevant reasoning behaviors, which are difficult to discover through reinforcement learning alone (Yue et al., 2025).

We then switch to reinforcement learning with output correctness as the reward. However, the I/O-based rewards only evaluate the final output and fail to assess intermediate execution steps. To overcome this, we introduce *white-box reward signals*, which generate verifiable questions from execution traces and reward the model based on its predictions of control flow and variable states, including values and types.

Trace collection from the interpreter. Given a synthesized program f and input x, we execute $f(x)$ using an interpreter to obtain an execution trace $\tau = (l_t, \sigma_t)_{t=1}^{T}$, where l_t is the executed statement at step t, and σ_t is the program state, including values and types of in-scope variables.

White-box question construction. From the execution trace τ, we deterministically construct two types of white-box questions: (i) *Control-flow questions*, which ask the model to predict the next executed statement l_{t+1}; (ii) *Data-flow questions*, which ask the model to predict updated variable values and types in σ_{t+1}. All questions are generated automatically from the interpreter and have a unique verifiable answer derived from the trace (see Appendix B.3 for details).

White-box reward function. We design the white-box reward function as follows:

$$R_{\text{white-box}} = 2 \cdot \Big((1-\alpha)\, R^{(\text{I}\to\text{O})} + \alpha\, R_{\text{white}} \Big),$$

where $\alpha \in [0, 1]$ balances the weight of final I/O correctness and white-box execution accuracy, and the factor of 2 ensures that the overall reward value ranges from 0 to 2. The term $R^{(\text{I}\to\text{O})}$ measures correctness under the input–output prediction setting and is a binary reward that takes value 1 if the model's predicted output matches the ground-truth output, and 0 otherwise. The term R_{white} measures the model's accuracy in predicting intermediate execution states and is computed over a sampled set Q_s of white-box questions:

$$R_{\text{white}} = \frac{1}{|Q_s|} \sum_{q_j \in Q_s} \mathbb{I}[a_j = a_j^*],$$

where a_j is the model's answer to question q_j, and a_j^* is the corresponding ground-truth answer derived from execution traces. This term reflects the model's accuracy in predicting intermediate execution states.

O→I prediction reward function. To encourage the model to reason in both directions and reduce reliance on forward input-to-output pattern matching, we also include reverse prediction tasks where the model predicts inputs from outputs. Since one output may have multiple valid inputs, we do not define white-box questions in this case. Instead, we assign a reward of 2 if the predicted input produces the correct output when executed, and 0 otherwise, maintaining the same $[0, 2]$ reward scale for consistency.

2.2.2 Stage II: RL for code generation

Once the model has obtained the code reasoning ability in the first stage, we further post-train it for the code generation task. The goal of this stage is to align the model's execution reasoning abilities with the objective of generating functionally correct programs. Following the previous study (Cui et al., 2025), we use a reward $R^{(gen)}$ defined as the proportion of unit tests the generated solution successfully passes:

$$R^{(gen)} = \frac{\text{Number of passed tests}}{\text{Total number of tests}}$$

This reward signal guides the model to apply its execution reasoning capabilities to generate functionally correct code.

3 Experiment Setup

3.1 Training Details

We use Qwen2.5-Coder-7B-Instruct (Hui et al., 2024) as the base model. We perform full-parameter SFT for the warm-up stage, and then apply GRPO (Guo et al., 2025) for both Stage I and Stage II. We use a maximum sequence length of 4096 for training, and for RL we sample $n = 8$ rollouts per prompt for 500 steps with a KL coefficient of 0.0. Full SFT and RL configurations are provided in Appendix C.1 and Appendix C.2. For Stage I, we use 30K synthesized samples for the SFT warm-up and another 30K for white-box RL. For each RL instance, we sample up to 10 white-box questions to compute R_{white} (see full setup in Appendix E.1). We set $\alpha = 0.5$ to balance terminal I/O reward and step-level white-box accuracy.

Varying α in $\{0.25, 0.5, 0.75\}$ yields similar performance (see Appendix D.2). For code-generation RL, we use the PrimeCode (Cui et al., 2025) dataset, sourced from APPS (Hendrycks et al., 2021), CodeContests (Li et al., 2022), TACO (Li et al., 2023), and CodeForces (Penedo et al., 2025).

3.2 Benchmarks

For code reasoning, we evaluate on three widely used benchmarks: CRUXEval (Gu et al., 2024b), LiveCodeBench-Exec (Jain et al., 2024), and REval (Chen et al., 2024), following the settings of prior work (Li et al., 2025; Ding et al., 2024a; Chen et al., 2025). The REval benchmark evaluates whether the model can correctly infer control flow, variable values, and variable types during the execution process. For code generation, we evaluate on three standard benchmarks: EvalPlus (Liu et al., 2023c), LiveCodeBench-V6 (Jain et al., 2024), and BigCodeBench (Zhuo et al., 2024). All evaluations use greedy sampling with the temperature set to 0.0, and we report pass@1 as the evaluation metric.

3.3 Baselines

For code reasoning, we compare our model against strong large-sized LLMs, including Qwen2.5-Coder-32B-Instruct (Hui et al., 2024) and Llama3-Instruct-70B (Dubey et al., 2024). We additionally include SEMCODER (Ding et al., 2024a) and CODEI/O (Li et al., 2025), both tuned on Qwen2.5-Coder-7B-Instruct. For code generation, we include larger models like Llama3-Instruct-70B (Dubey et al., 2024), DeepSeek-Coder-V2-Lite-Instruct (Zhu et al., 2024), and Qwen2.5-Coder-14B-Instruct (Hui et al., 2024) as baselines. SEMCODER (Ding et al., 2024a) and CODEI/O (Li et al., 2025) are also evaluated on code generation benchmarks for completeness.

4 Experimental Results

4.1 Code Reasoning Results

Our SFT + white-box RL model outperforms strong baseline models. Both SFT and white-box RL are effective. Table 1 summarizes our experimental results on CRUXEval, LiveCodeBench-Exec, and REval. Across comparable 7B variants trained on the same synthesized corpus (details in Appendix E.1), I/O RL substantially improves over our base model Qwen2.5-Coder-7B-Instruct, adding SFT warm-up yields further gains, and replacing I/O RL with white-box RL achieves the best overall performance. As shown in the last column, our final model improves the average score from 60.8 to 80.8 (+20.0) and is competitive with Qwen2.5-Coder-32B-Instruct (77.9).

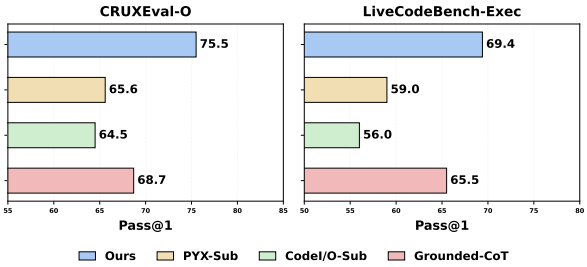

Figure 3: Data efficiency comparison at a fixed training scale (15K examples). We report Pass@1 on CRUXEval-O and LiveCodeBench-Exec for models fine-tuned with different datasets.

4.2 Code Generation Results

Two-stage RL is effective. Table 2 presents code generation results on HumanEval, MBPP, LiveCodeBench, and BigCodeBench. We further train the models in Table 1 with unit-test RL (recall in Section 2.2.2). While single-stage GRPO (UT RL) already improves the base 7B model (53.9), initializing GRPO from a reasoning-enhanced checkpoint yields consistently better performance (see last three rows). Our best two-stage variant (SFT + white-box RL + unit-test GRPO) achieves the highest average score (57.1) and improves pass@1 by up to 5.9 points over the pure-GRPO baseline.

Code generation benefits from white-box reinforcement learning. Among models trained with unit-test RL, the progression from UT-RL (53.9) to I/O RL + UT-RL (54.6), SFT + I/O RL + UT-RL (54.9), and finally SFT + white-box RL + UT-RL (57.1) shows a consistent upward trend. These results indicate that the fine-grained execution knowledge learned via white-box RL, such as tracking control flow and variable states, not only boosts code reasoning performance but also transfers to realistic code generation tasks.

4.3 Data Efficiency

To isolate the impact of data quality, we compare our synthesized data with three representative datasets: PYX-Sub (Ding et al., 2024a), CodeI/O-Sub (Li et al., 2025), and Grounded-CoT (Jung et al., 2025), which correspond to the two execution-supervision paradigms in Section 1 (I/O CoT vs. LLM-translated traces). We sample

Table 1: Code reasoning experimental results on CRUXEval, LiveCodeBench, and REval. Average is computed over all fine-grained metrics.

Model	Size	CRUXEval		LiveCodeBench-Exec	REval				Average
		CXEval-O	CXEval-I	LCB-O	Coverage	State	Path	Output	
Qwen2.5-Coder-Instruct	32B	78.2	83.4	80.6	84.6	66.7	68.7	83.3	77.9
LLaMA3-Instruct	70B	63.7	61.3	56.4	85.3	59.2	40.3	74.6	63.0
SemCoder	7B	66.7	65.3	58.9	80.8	54.7	50.7	65.3	63.2
CODEI/O	7B	62.5	67.9	60.8	70.9	52.5	45.9	60.8	60.1
Qwen2.5-Coder-Instruct	7B	61.0	66.0	58.0	78.6	51.7	49.7	60.3	60.8
+ I/O O/I RL	7B	74.9	76.5	72.2	83.2	65.1	50.4	77.3	71.4
+ SFT + I/O O/I RL	7B	83.5	**84.3**	81.0	84.4	67.2	51.4	82.3	76.3
+ SFT + white-box RL	7B	**85.6**	81.0	**82.3**	**85.8**	**74.5**	**73.0**	**83.2**	**80.8**

Table 2: Code generation results on HumanEval, MBPP, LiveCodeBench, and BigCodeBench. Average is computed over all fine-grained metrics.

Backbone	Size	HumanEval		MBPP		LiveCodeBench			BigCodeBench		Average
		HE	HE+	MBPP	MBPP+	Easy	Medium	Hard	Full	Hard	
Qwen2.5-Coder-Instruct	32B	92.7	87.2	90.2	75.1	84.5	22.0	3.4	49.6	27.0	59.0
Qwen2.5-Coder-Instruct	14B	89.6	**87.2**	86.2	72.8	71.0	13.5	2.5	48.4	22.2	54.8
DS-Coder-V2-Lite-Inst.	16B	81.1	75.6	82.8	70.4	61.2	9.6	3.1	36.8	16.2	48.5
LLaMA3-Instruct	70B	77.4	72.0	82.3	69.0	61.0	9.6	3.2	**54.5**	**27.0**	50.7
SemCoder	7B	88.6	83.8	85.9	71.0	61.8	12.8	2.9	42.3	19.5	52.1
CODEI/O	7B	86.0	80.5	81.0	69.4	56.5	7.2	2.9	40.5	17.2	49.0
Qwen2.5-Coder-Instruct	7B	88.4	84.1	83.5	71.7	60.0	9.2	3.0	41.0	18.2	51.0
+ UT RL	7B	87.2	82.3	85.3	72.1	64.4	13.5	4.1	49.3	23.0	53.9
+ I/O RL + UT RL	7B	90.2	82.9	85.2	71.2	68.2	15.6	3.9	**50.6**	23.3	54.6
+ SFT + I/O RL + UT RL	7B	**91.5**	84.0	86.1	72.3	69.5	14.4	3.9	49.6	23.1	54.9
+ SFT + white-box RL+ UT RL	7B	90.9	84.8	**89.4**	**75.1**	**74.5**	**18.4**	**5.9**	49.4	**25.7**	**57.1**

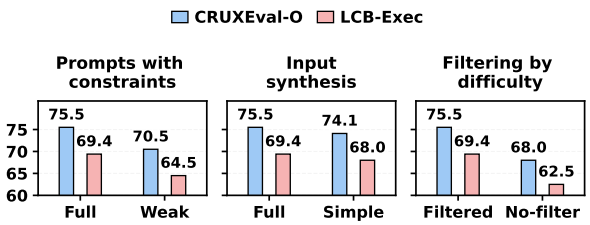

Figure 4: Ablation study on our synthesis pipeline on CRUXEval-O and LiveCodeBench-Exec. We report pass@1 on models finetuned with different data synthesis variants.

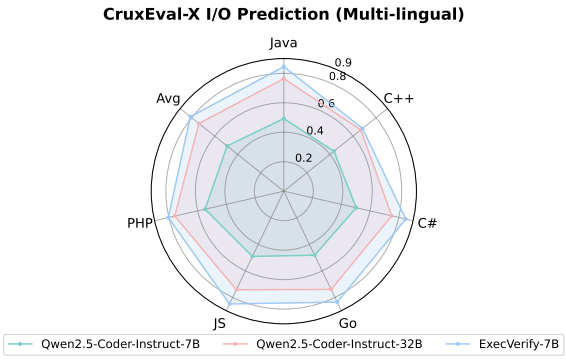

Figure 5: CRUXEval-X Multilingual I/O Prediction: Comparison with Qwen2.5-Coder-Instruct (7B/32B).

15K instances randomly from each training dataset (see full setup in Appendix E.2). As shown in Figure 3, our dataset achieves the highest performance on both CRUXEval-O and LiveCodeBench-Exec, demonstrating superior data efficiency.

4.4 Ablation on Data Synthesis

Our synthesis pipeline has three key components: (i) generating prompts with structural constraints, (ii) input synthesis, and (iii) filtering by difficulty. To isolate their contributions, we run three SFT-only ablations on the I/O prediction task using 15K training examples (see Appendix E.3 for all configurations). Figure 4 shows that each component improves pass@1 on both benchmarks.

4.5 Cross-language Generalization

CRUXEval-X (Xu et al., 2025a) is a multilingual code execution reasoning benchmark. To test whether our execution reasoning improvements extend beyond Python, we evaluate I/O prediction on CRUXEval-X across six programming lan-

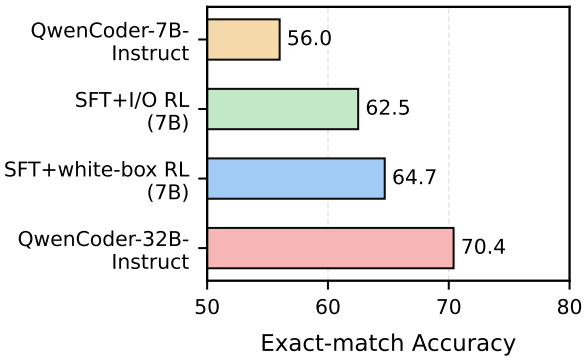

Figure 6: Experimental results on Library-involved I/O Prediction

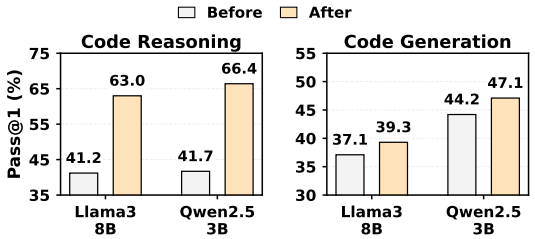

Figure 7: Averaged performance metrics on Code Reasoning and Generation benchmarks. The results demonstrate the generalization of our method across various model sizes and architectures.

guages (Java, C++, C#, Go, JavaScript, and PHP). "Avg" denotes the average accuracy across these six languages. As shown in Figure 5, *ExecVerify*-7B consistently outperforms the same-family base model Qwen2.5-Coder-7B-Instruct across all languages, and is also competitive with the much larger Qwen2.5-Coder-32B-Instruct. These results show that the execution reasoning ability transfers effectively across programming languages.

4.6 Library-involved I/O Prediction

To evaluate *ExecVerify* on code that depends on external libraries, we construct a library-involved I/O prediction benchmark from BigCodeBench. For each task, we extract an input–output pair from the interactive example in the task description and use the task's canonical solution as the executable source program. At evaluation time, we provide the extracted input and ask the model to predict the output of the canonical solution. We report the exact match accuracy. To ensure determinism, we filter out tasks involving randomness or external resources (e.g., random number generation, file I/O), resulting in 241 test cases (see Appendix E.4).

As shown in Figure 6, Qwen2.5-Coder-7B-Instruct achieves 56.0, SFT+I/O RL improves to 62.5, and SFT+WHITE-BOX RL further reaches 64.7, compared to 70.4 from the much larger Qwen2.5-Coder-32B-Instruct. These results indicate that our improvements transfer to library-involved code settings.

4.7 Ablations on White-box Questions

We ablate the two types of white-box questions by scoring either only control-flow questions (CF-only) or only data-flow questions (DF-only), while keeping the prompting format unchanged (i.e., unscored questions are still generated). Table 3 shows that the two signals are complementary: CF-only improves control-flow metrics but reduces state accuracy, while DF-only enhances variable state prediction but hurts CF performance. Scoring both types (Full) yields the best overall balance and the highest average score.

4.8 Training Dynamics

We additionally report training dynamics in Figure 16. Stage I white-box RL provides stable improvements in both white-box accuracy and I/O prediction. Stage II consistently outperforms training the base model from scratch on code generation across the entire training process. Detailed analysis is provided in Appendix D.1.

4.9 Generalization across Model Sizes and Architectures

We further assess the generalizability of our approach across model sizes and architectures by applying the same training pipeline to Qwen2.5-Coder-3B-Instruct (Hui et al., 2024) and Llama3-8B-Instruct (Dubey et al., 2024). As shown in Figure 7, our method yields consistent improvements on Code Reasoning and Code Generation evaluations, indicating that the proposed pipeline transfers beyond a single model family and supports robust gains across different LLM architectures.

5 Related Work

Enhance LLM's Performance on Code Execution Reasoning Previous works (Liu et al., 2023a; Ding et al., 2024b) fine-tune LLMs such as UnixCoder (Guo et al., 2022) directly on raw execution traces. Self-Debugging (Chen et al., 2023) further finds that directly feeding raw traces can even undermine LLM's performance on program repair. To alleviate this, Next (Ni et al., 2024) injects execution information into debugging comments when

Table 3: Control-flow vs. data-flow ablations for Stage I white-box RL. Best results in each column are highlighted in bold. Average is the mean over all metrics.

Model	CRUXEval		LiveCodeBench	REval				Average
	CXEval-O	CXEval-I	LCB-O	Coverage	State	Path	Output	
Full	**85.6**	81.0	**82.3**	85.8	74.5	73.0	**83.2**	**80.8**
CF-only	84.1	**82.6**	80.9	**86.0**	68.8	**74.7**	82.5	79.9
DF-only	84.9	82.1	**82.3**	84.9	**75.7**	54.6	82.1	78.1

fine-tuning the model. More recent works adopt other training paradigms. SemCoder (Ding et al., 2024a) and CODEIO (Li et al., 2025) fine-tune models on input–output and output–input reasoning chains extracted from stronger teacher models. TracePile (Chen et al., 2025) and Code Execution as Grounded Supervision (Jung et al., 2025) also rely on a teacher LLM to translate execution traces into natural-language and perform supervised fine-tuning on it. In contrast, *ExecVerify* adopts a new training paradigm: it converts execution traces into white-box questions about control flow and data flow. Therefore, it provides dense and verifiable rewards for intermediate execution steps throughout reinforcement learning.

Evaluating LLMs on Code Execution Reasoning Early works (Liu et al., 2023a; Ding et al., 2024b) evaluate trained models on their own collected datasets. More recently, CRUXEval (Gu et al., 2024b) was proposed as a public benchmark for code execution reasoning, and CRUXEval-X (Xu et al., 2025a) extends it to multiple programming languages. REval (Chen et al., 2024) further refines the prediction task by requiring LLMs to predict intermediate execution states rather than only final outputs and CORE (Xie et al., 2025) evaluates LLMs on more complex static analysis tasks.

Data Synthesis for Code LLMs Researchers have proposed multiple synthesized datasets for training code generation models, such as CodeAlpaca (Chaudhary, 2023), Evol-Instruct-Code (Luo et al., 2023), OSS-Instruct (Wei et al., 2023), PackageInstruct (Huang et al., 2025), and KODCODE (Xu et al., 2025b). For code execution reasoning, existing works (Ding et al., 2024a; Li et al., 2025; Chen et al., 2025; Jung et al., 2025) typically passively mine or generate code from real-world code snippets. In contrast, *ExecVerify* actively synthesizes data by applying structural constraints and difficulty filtering, resulting in higher-quality data (see Section 2.1).

6 Conclusion and Future Work

In this work, we presented *ExecVerify*, a post-training framework for teaching code LLMs to reason about program execution. On the data side, we build a constraint-based synthesis pipeline that actively generates executable programs under structural constraints, augments them with diverse inputs, and applies difficulty-aware filtering to form a curriculum-style dataset containing challenging yet solvable instances. On the learning side, we propose a two-stage training pipeline: Stage I focuses on execution reasoning via white-box reinforcement learning, rewarding the model for answering verifiable questions about intermediate control flow and variable states, and Stage II adapts the model to code generation with unit-test–based rewards. Experiments show that a 7B code model trained with *ExecVerify* achieves performance competitive with 32B models on execution reasoning benchmarks, and demonstrates improvement over post-training baselines on standard code generation benchmarks.

In future work, we plan to extend *ExecVerify* along three directions. First, on the data side, we will broaden the coverage of types and methods in our synthesis pipeline and include more libraries. Second, we aim to generalize our framework to other programming languages. Finally, we intend to move from function-level snippets to project-level code and model the execution process of multi-file and project-level programs.

7 Limitations

Our synthesized data is currently restricted to Python and mainly covers built-in types with limited library usage, so real-world code coverage is incomplete. Our experiments are conducted at the function or snippet level rather than on multi-file or project-level code. In addition, the proposed white-box reinforcement learning pipeline incurs higher computational and engineering overhead than standard supervised fine-tuning.

References

Sahil Chaudhary. 2023. Code alpaca: An instruction-following llama model for code generation.

Junkai Chen, Zhiyuan Pan, Xing Hu, Zhenhao Li, Ge Li, and Xin Xia. 2024. Reasoning runtime behavior of a program with llm: How far are we? *arXiv preprint arXiv:2403.16437*.

Nuo Chen, Zehua Li, Keqin Bao, Junyang Lin, and Dayiheng Liu. 2025. Chain of execution supervision promotes general reasoning in large language models. *arXiv preprint arXiv:2510.23629*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.

Ganqu Cui, Lifan Yuan, Zefan Wang, Hanbin Wang, Wendi Li, Bingxiang He, Yuchen Fan, Tianyu Yu, Qixin Xu, Weize Chen, and 1 others. 2025. Process reinforcement through implicit rewards. *arXiv preprint arXiv:2502.01456*.

Yangruibo Ding, Jinjun Peng, Marcus Min, Gail Kaiser, Junfeng Yang, and Baishakhi Ray. 2024a. Semcoder: Training code language models with comprehensive semantics reasoning. *Advances in Neural Information Processing Systems*, 37:60275–60308.

Yangruibo Ding, Benjamin Steenhoek, Kexin Pei, Gail Kaiser, Wei Le, and Baishakhi Ray. 2024b. Traced: Execution-aware pre-training for source code. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, and 1 others. 2024. The llama 3 herd of models. *arXiv e-prints*, pages arXiv–2407.

Alex Gu, Wen-Ding Li, Naman Jain, Theo Olausson, Celine Lee, Koushik Sen, and Armando Solar-Lezama. 2024a. The counterfeit conundrum: Can code language models grasp the nuances of their incorrect generations? In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 74–117.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I Wang. 2024b. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Sonam Gupta, Yatin Nandwani, Asaf Yehudai, Dinesh Khandelwal, Dinesh Raghu, and Sachindra Joshi. 2025. Selective self-to-supervised fine-tuning for generalization in large language models. *arXiv preprint arXiv:2502.08130*.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and 1 others. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

Siming Huang, Tianhao Cheng, Jason Klein Liu, Weidi Xu, Jiaran Hao, Liuyihan Song, Yang Xu, Jian Yang, Jiaheng Liu, Chenchen Zhang, and 1 others. 2025. Opencoder: The open cookbook for top-tier code large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 33167–33193.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Rasha Ahmad Husein, Hala Aburajouh, and Cagatay Catal. 2025. Large language models for code completion: A systematic literature review. *Computer Standards & Interfaces*, 92:103917.

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

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*.

Dongwon Jung, Wenxuan Zhou, and Muhao Chen. 2025. Code execution as grounded supervision for llm reasoning. *arXiv preprint arXiv:2506.10343*.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, and 1 others. 2022. The stack: 3 tb of permissively licensed source code. *arXiv preprint arXiv:2211.15533*.

Junlong Li, Daya Guo, Dejian Yang, Runxin Xu, Yu Wu, and Junxian He. 2025. Codei/o: Condensing reasoning patterns via code input-output prediction. *arXiv preprint arXiv:2502.07316*.

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

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, and 1 others. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Chenxiao Liu, Shuai Lu, Weizhu Chen, Daxin Jiang, Alexey Svyatkovskiy, Shengyu Fu, Neel Sundaresan, and Nan Duan. 2023a. Code execution with pre-trained language models. *arXiv preprint arXiv:2305.05383*.

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

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023c. 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*.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

Ansong Ni, Miltiadis Allamanis, Arman Cohan, Yinlin Deng, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2024. Next: Teaching large language models to reason about code execution. *arXiv preprint arXiv:2404.14662*.

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

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2024. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv: 2409.19256*.

Zafir Stojanovski, Oliver Stanley, Joe Sharratt, Richard Jones, Abdulhakeem Adefioye, Jean Kaddour, and Andreas Köpf. 2025. Reasoning gym: Reasoning environments for reinforcement learning with verifiable rewards. *arXiv preprint arXiv:2505.24760*.

Qwen Team. 2024. Qwq: Reflect deeply on the boundaries of the unknown. *Hugging Face*.

Yihan Wang, Si Si, Daliang Li, Michal Lukasik, Felix Yu, Cho-Jui Hsieh, Inderjit S Dhillon, and Sanjiv Kumar. 2022. Two-stage llm fine-tuning with less specialization and more generalization. *arXiv preprint arXiv:2211.00635*.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Empowering code generation with oss-instruct. *arXiv preprint arXiv:2312.02120*.

Danning Xie, Mingwei Zheng, Xuwei Liu, Jiannan Wang, Chengpeng Wang, Lin Tan, and Xiangyu Zhang. 2025. Core: Benchmarking llms code reasoning capabilities through static analysis tasks. *arXiv preprint arXiv:2507.05269*.

Ruiyang Xu, Jialun Cao, Yaojie Lu, Ming Wen, Hongyu Lin, Xianpei Han, Ben He, Shing-Chi Cheung, and Le Sun. 2025a. Cruxeval-x: A benchmark for multilingual code reasoning, understanding and execution. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 23762–23779.

Zhangchen Xu, Yang Liu, Yueqin Yin, Mingyuan Zhou, and Radha Poovendran. 2025b. Kodcode: A diverse, challenging, and verifiable synthetic dataset for coding. *arXiv preprint arXiv:2503.02951*.

He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural program repair with execution-based backpropagation. In *Proceedings of the 44th international conference on software engineering*, pages 1506–1518.

Yang Yue, Zhiqi Chen, Rui Lu, Andrew Zhao, Zhaokai Wang, Shiji Song, and Gao Huang. 2025. Does reinforcement learning really incentivize reasoning capacity in llms beyond the base model? *arXiv preprint arXiv:2504.13837*.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. 2024. Llamafactory: Unified efficient fine-tuning of 100+ language models. *arXiv preprint arXiv:2403.13372*.

Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, and 1 others. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, and 1 others. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*.

A Dataset Statistics and Analysis

A.1 Difficulty Imbalance in Existing Execution Training Datasets

To better understand the existing execution-style training datasets, we conducted a small-scale empirical study on two widely used training datasets from SEMCODER (Ding et al., 2024a) and CODEIO (Li et al., 2025).

On a random sample of 15k test cases from SEMCODER, the Qwen2.5-Coder-Instruct-7B model already solves roughly 70% of problems in a single pass@1 attempt, without any additional reasoning-specific fine-tuning. This suggests that a large portion of SEMCODER is trivial for modern code LLMs and provides limited signal for improving execution reasoning.

In contrast, when we randomly sample 15k problems from the training dataset CODEIO, we observe the opposite phenomenon: even the strong model qwq32b (Team, 2024) frequently fails to find any solution. A significant subset of 52.1% CODEIO instances is unsolved by current frontier models. Independent evidence from REASONING GYM (Stojanovski et al., 2025) further supports this picture: in their dataset, the CodeIO programs are explicitly configured as high-difficulty "code" tasks, and the reported zero-shot accuracies of strong reasoning models like QwQ-32B on these tasks remain low even under the easy settings.

A.2 Difficulty and Complexity Distribution

Figure 8 shows the difficulty distribution of our synthesized problems, measured by the number of successful trials obtained by a baseline code model Qwen2.5-Coder-7B-Instruct on the raw and mutated datasets. For each problem, we run the model with temperature 1.0 with the task input-output prediction for ten independent trials and record the number of trials $k \in \{0, \ldots, 10\}$ whose predictions are correct. The histogram indicates that both datasets cover a wide range of difficulty levels, and that input mutation slightly shifts probability mass from trivially easy problems toward harder ones while preserving overall diversity.

Table 4 reports several structural complexity metrics computed over the final difficulty-filtered dataset. On average, each snippet contains 9.93 non-empty, non-comment lines of code (LOC), with a median of 9. The maximum depth of the full Python abstract syntax tree (AST) has a mean of 9.74 and a median of 10, reflecting non-trivial expression structure even for relatively short snippets. Each snippet includes on average 1.43 branch constructs (e.g., `if`/`elif`/ternary expressions) and 0.85 loop constructs (e.g., `for`/`while` and comprehensions), both with medians of 1.

Metric	Mean	Median
Lines of code (LOC)	9.93	9
AST depth (full syntax tree)	9.74	10
#branches (if / elif / ternary)	1.43	1
#loops (for / while / comp.)	0.85	1
Control-flow nesting depth	2.86	3

Table 4: Code complexity statistics of the final difficulty-filtered dataset. AST depth is measured as the maximum depth of the full Python abstract syntax tree, while control-flow nesting depth counts only structured blocks such as `if`/`for`/`while`/`try`/`with`/function and class definitions.

To better characterize control flow, we additionally measure the nesting depth of structured blocks, counting only `if`, `for`, `while`, `try`, `with`, function definitions, and class definitions. The resulting control-flow nesting depth has a mean of 2.86 and a median of 3, indicating that most instances involve multiple layers of nested logic rather than flat scripts.

A.3 Type Distribution

Figure 9 presents the distribution of Python built-in types and related operations in the final difficulty-filtered dataset. String-manipulation problems (`str`) constitute nearly half of all instances, followed by sets (`set`), lists (`list`), and dictionaries (`dict`). There are also less frequent but still tasks such as tuples, floating-point numbers, and operations such as `zip`, `enumerate`, `reverse`, `range`, and `filter`. This mix of frequent and long-tail types ensures that the model is exposed to a broad spectrum of everyday Python programming primitives.

A.4 Filtering Statistics

Figure 10 summarizes the effect of our filtering stages on both the raw and mutated datasets. Starting from 239,992 raw samples and 239,466 mutated samples, execution-based filtering removes snippets that fail to run successfully or violate basic output constraints (e.g., runtime exceptions, timeouts, or excessively long outputs), leaving 201,537 raw and 191,463 mutated samples. We then apply difficulty-based filtering using the success-count distribution described in Section 2.1.2, retaining

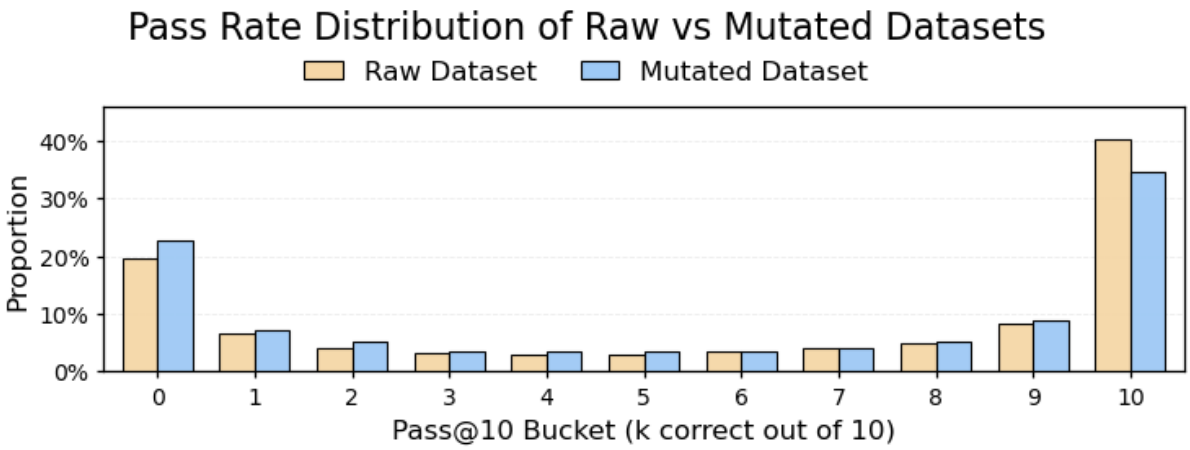

Figure 8: Difficulty distribution of the raw and mutated datasets, measured by the number of successful trials k out of 10 for each synthesized problem.

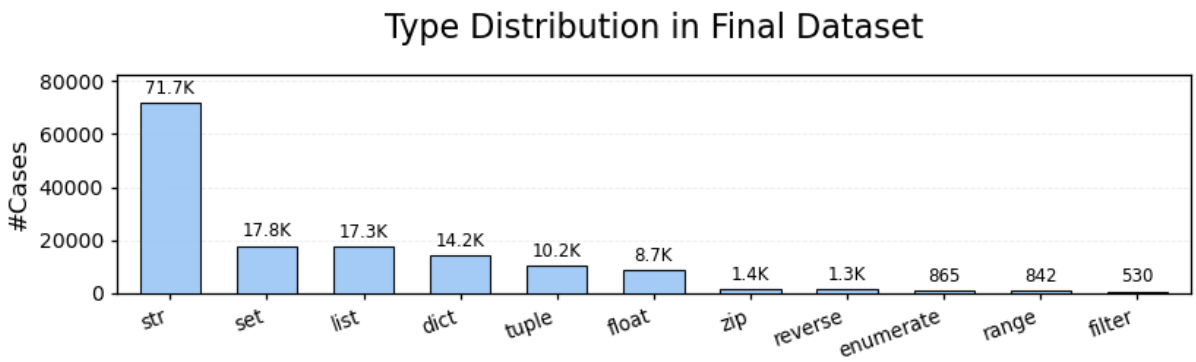

Figure 9: Distribution of Python built-in types in the final difficulty-filtered dataset.

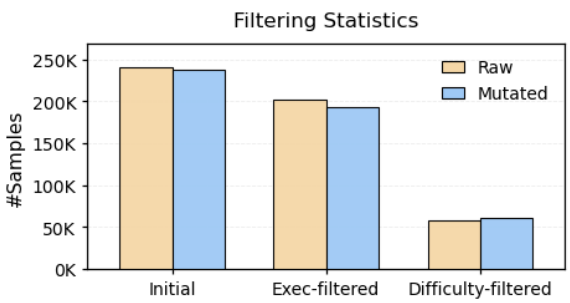

Figure 10: Filtering statistics of the synthesized raw and mutated datasets, showing the number of samples that remain after execution-based and difficulty-based filtering stages.

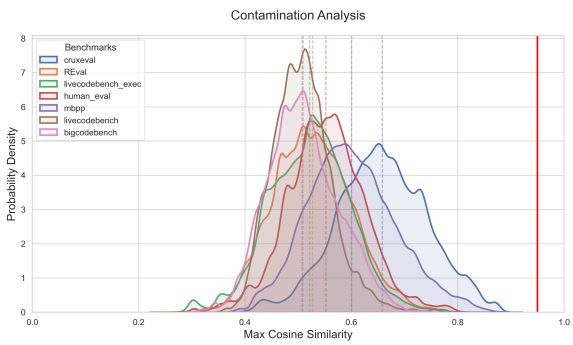

Figure 11: Contamination analysis between our synthesized dataset and downstream evaluation benchmarks, showing the distribution of maximum cosine similarity scores per training instance and the 0.95 threshold used to flag potential overlaps.

119,358 instances that are non-trivial for the baseline model. The decrease across stages illustrates how each component of the pipeline progressively improves data quality while preserving a large and diverse training set.

A.5 Contamination Analysis

We evaluate potential contamination between our synthesized training data and existing evaluation benchmarks following the same embedding-based protocol as KODCODE. For each question in our

dataset, we encode its natural-language description using the `all-mpnet-base-v2` sentence-embedding model, and apply the same encoder to all problems from our downstream benchmarks. We then compute cosine similarity between each training instance and all benchmark questions, and record the maximum similarity for each instance. Following prior work, we adopt 0.95 as a conservative similarity threshold for flagging potential contamination. Figure 11 shows the distribution of maximum cosine similarity scores across all training instances, together with a vertical line at 0.95. In our data, **no training instance exceeds this threshold**, and the entire distribution lies clearly below 0.95, suggesting that near-duplicates or paraphrased copies of benchmark items are extremely rare. We additionally perform a manual inspection of the few highest-similarity pairs (e.g., instances with similarity close to the right tail of the distribution) and confirm that they differ substantially in both surface form and semantics. Consequently, we do not remove any training examples at this stage and consider our synthesized dataset to be effectively contamination-free with respect to the benchmarks used in our evaluation.

B Data Synthesis Details

This section provides the specific prompt templates and logical details used in the *Constraint-Based Data Synthesis* pipeline described in Section 2.1, to support the reproducibility of our experiments.

B.1 Code Synthesis Prompts with Constraints

To synthesize executable programs that exhibit non-trivial execution behavior, we use QWQ-32B as the generator model with explicit structural constraints. Figure 12 shows a representative prompt template used for generating Python code that tests the `rstrip` method of the `str` type. The system message positions the model as an expert Python programmer and instructs it to strictly adhere to the given constraints.

The user prompt specifies: (i) *control-structure constraints*, such as the requirement to include a `for` loop and to nest an `if` statement inside a `while` loop; (ii) *method-call constraints*, such as invoking the target method multiple times and combining it with at least one additional built-in method; and (iii) formatting requirements, such as avoiding comments and emitting a single Markdown code block. By enforcing such constraints at generation time, we obtain programs that naturally contain nested control flow and rich interactions among multiple built-in operations.

Curriculum levels. To encourage the model to gradually acquire more complex execution patterns, we organize our constraint-based prompts into three curriculum levels based on the required control flow and method interactions. Table 5 summarizes the design.

B.2 Input Synthesis and Mutation

Given a synthesized program, we create diverse inputs to probe its execution behavior. We first use the same QWQ-32B generator to construct an input. We then perform type-aware input mutation to obtain valid samples with mutated inputs.

Figure 13 presents the prompt used for input mutation. The template exposes the original code snippet and asks the model to directly rewrite the arguments of the entry-point function call, while respecting a list of reference values (e.g., candidate integers and strings) and a set of mutation guidelines. These guidelines instruct the model to, for example, increase the length of strings or containers and to modify arguments in a way that remains consistent with the reference values and program semantics. The output is again required to be a single Markdown code block that calls the entry function and prints the result.

Figure 14 shows an example of the resulting mutation. The top part displays the original code together with its initial assertion, while the bottom part shows the mutated version produced by our procedure. The highlighted assertion demonstrates how the input string is replaced by a synthetic value that changes the execution trace yet still exercises the same functionality.

B.3 White-Box Question Generation

To obtain supervision that directly targets execution reasoning, we convert each instrumented program run into a set of *white-box questions* derived from its execution trace. Concretely, we execute the synthesized Python code in a sandbox and use the built-in `traceback` facility to record the sequence of executed statements together with the evolving program state.

Given an execution trace, we construct two types of questions:

- **Variable-state (data-flow) questions.** For each executed statement, we compare the val-

Level	Structural constraints	Share (%)
1	Single built-in method call without any control-flow constructs.	20.9
2	Multiple method calls with at least one shallow `if` or `for` statement, but no nested control-flow blocks.	14.2
3	Multiple method calls with nested `if`/`for`/`while` statements, and a maximum control-flow nesting depth of 3.	64.9

Table 5: Curriculum levels used in constraint-based code synthesis, grouped by the required control-flow structure and method interactions in the final dataset.

An example of prompt with constraints

```
System Prompt:  You are an expert Python programmer.  Your task is to
generate valid, executable Python code that strictly adheres to a complex
set of structural and logical constraints.

---

User Prompt:
Task:  Write Python code that tests the rstrip method of the str type.
Constraints:
 • Control Structure Constraints:
   - The code must include a for statement.
   - You must nest an if statement inside the for loop.
 • Method Call Constraints:
   - Invoke the test method multiple times within the code.
   - Integrate at least one additional built-in method (preferably from str,
     list, set, or tuple).
Formatting:
 • Do not include any comments.
 • Call the entry-point function exactly once.
Output Format:
You must output a markdown code block:
```python
complete code snippet
```
```

Figure 12: A constraint-based code synthesis prompt used for generating Python programs that test specific built-in methods while strictly adhering to structural and formatting requirements.

ues of all in-scope variables immediately before and after the statement. Whenever a variable changes, we create a question that asks for its value and Python type after the statement has executed. These questions encourage the model to track how data flows through the program.

- **Next-statement (control-flow) questions.** For each executed statement, we inspect the next statement in the trace. If the current statement is a control-flow construct such as `if`, `while`, or `for`, we create a question asking for the exact source line that will be executed next. In addition, whenever the line number of the next executed statement is smaller than that of the current one (i.e., control transfers backwards in the source file, as in loop iterations or taken branches), we also generate a next-statement question. These questions require the model to reason about branch conditions and loop behavior.

All candidate questions from a trace are collected into a problem set for the corresponding program. For each training instance, we shuffle this set and sample up to ten questions. Some traces produce fewer than ten valid questions, so the resulting number of white-box questions per instance varies. On average, each program contributes 7.8 white-box questions, of which 3.2 are control-flow (next-statement) questions and 4.6 are data-flow (variable-state) questions. This yields dense supervision over both control-flow and value-tracking

An example of prompt for input mutation

Task: Modify the arguments in the provided code based on specific constraints.

Provided Code:

```
def test_rstrip(s):
    result = s.rstrip()
    for char in result:
      if char.isalpha():
        result = result.lstrip(char)
      elif char.isdigit():
        result = result.strip(char)
      else:
        result = result.rstrip(char)
    return result
assert test_rstrip(" hello world ") == " hello world"
```

Instructions:
- Reconstruct the arguments in the function call by **directly writing the arguments** inside the function call (do not assign them to variables first).
- You must use **one or more** values from the reference list below.

Reference Values:
- **Integer Values:** 12, 7, 11, 9, 11, 7, 14
- **String Values:** 'EEFujAr', 'E2NC97aoEt', 'ZWRus3xdc8', 'a-s*?Rx&;', 'STxJPmNuB4', 'TK07iWVF0', '%%%%%%%%%%%%'

Notes & Constraints:
- For types such as str, dict, list, or set, try to increase their length.
- You may modify the code if needed, as long as it aligns with the reference values.

Output Format:
Call the entry point function with the new arguments and use the print statement to print the result. Return only a single Markdown code block:

```
  ```python
  # complete code snippet
  ```
```

Figure 13: An example of prompt for mutating inputs in the code

aspects of execution.

Figure 15 illustrates the prompt used to query the model with these white-box questions. The upper part displays the full code snippet with line numbers, while the lower part lists several next-statement and variable-state questions derived from a single execution trace, together with strict formatting rules for the model's reasoning and answers.

C Experimental Setup

C.1 Supervised Fine-Tuning (SFT)

Table 6 summarizes the hyper-parameters used in the supervised fine-tuning (SFT) stage. We fine-tune Qwen2.5-Coder-7B-Instruct in a full-parameter setting using the LLaMAFactory (Zheng et al., 2024) framework with DeepSpeed ZeRO-2 on a cluster of 8×H100 GPUs.

For SFT, we first randomly sample 30K examples from the dataset constructed in section 2.1 to form the `sft_new_dataset` split used for training. We apply the official `qwen` chat template and truncate each sequence to at most 4096 tokens. Data preprocessing uses 16 CPU workers and data loading uses 4 workers. Unless otherwise specified, all SFT experiments in this paper follow this configuration.

An example of Input Mutation

Code before input mutation:

```
def test_rstrip(s):
    result = s.rstrip()
    for char in result:
      if char.isalpha():
        result = result.lstrip(char)
      elif char.isdigit():
        result = result.strip(char)
      else:
        result = result.rstrip(char)
    return result
assert test_rstrip(" hello world ") == " hello world"
```

Code after input mutation:

```
def test_rstrip(s):
    result = s.rstrip()
    for char in result:
      if char.isalpha():
        result = result.lstrip(char)
      elif char.isdigit():
        result = result.strip(char)
      else:
        result = result.rstrip(char)
    return result
assert test_rstrip('E2NC97aoEt') == ""
```

Figure 14: An example of input mutation applied to the synthesized code that tests the `rstrip` method. The mutated input is highlighted.

Hyper-parameter	Value
Learning rate	1×10^{-5}
Number of epochs	2
Per-device batch size	2
Gradient accumulation steps	8
Effective batch size	128
Max sequence length	4096
Optimizer	AdamW
Learning rate scheduler	cosine
Warmup ratio	0.1
Precision	bfloat16
Fine-tuning type	full-parameter
Parallelism	DeepSpeed ZeRO-2

Table 6: Hyper-parameters for supervised fine-tuning (SFT).

C.2 Reinforcement Learning (GRPO)

Table 7 summarizes the hyper-parameters used in the GRPO-based reinforcement learning stages.

Stage I: reasoning RL. In the first RL stage, we start from the SFT checkpoint of **Qwen2.5-Coder-7B-Instruct** and apply GRPO on our synthetic execution-reasoning corpus. The reward combines input–output correctness and white-box signals (control-flow and data-flow questions derived from execution traces).

Stage II: code generation RL. In the second RL stage, we start from the Stage-I checkpoint and apply GRPO with the VERL (Sheng et al., 2024) framework on the PrimeCode (`eurus_prime`) train/validation splits.

Training is also performed on **8×H100** GPUs with FSDP (parameter and optimizer offloading) and gradient checkpointing enabled.

Unless otherwise specified, **both RL stages share the same GRPO hyper-parameters** as listed in Table 7; only the training data and reward functions differ between the two stages.

Hyper-parameter	Value
Algorithm	GRPO (VeRL)
Train batch size (prompts)	128
#responses per prompt n	8
Max prompt length	2048
Max response length	4096
Actor learning rate	1×10^{-6}
PPO mini-batch size	64
PPO micro-batch size / GPU	1
Rollout backend	vLLM
Tensor model parallel size	4
KL in loss / in reward	on / off (coef 0.0)
Total steps	500
Parallelism	FSDP (param & opt offload)
GPUs	8×H100

Table 7: Hyper-parameters for GRPO-based reinforcement learning (UT-RL) on the PrimeCode dataset.

D Additional Ablations and Training Dynamics

D.1 Training Dynamics

Figure 16 provides the training dynamics of Stage I and Stage II. In the Stage I curves, we observe that white-box RL yields stable gains on both white-box questions and I/O prediction. In the Stage II curve, we see that our two-stage training framework, which first trains the model for code reasoning and then trains it for code generation, consistently outperforms directly training the model for code generation from scratch, delivering stable improvements throughout training.

D.2 Sensitivity to the Reward Mixing Coefficient α

In Stage I (white-box RL), we combine the I/O-based reward and the white-box reward via a convex mixture:

$$r = (1-\alpha)\, r_{\text{I/O}} + \alpha\, r_{\text{WB}}, \qquad (1)$$

where α controls the relative weight of the white-box signal. In the main paper, we set $\alpha = 0.5$.

We evaluate $\alpha \in \{0.25, 0.5, 0.75\}$ while keeping all other training settings fixed. Table 8 reports CXEval-O, CXEval-I, LCB-O, and the fine-grained REval metrics (Coverage/State/Path/Output), along with the overall summary score. The experimental results indicate that performance is not sensitive to α within this range.

E Experimental Details

E.1 Variant setup for Table 1

All 7B variants in Table 1 are trained on the same pool of 60K programs randomly sampled from our synthesized reasoning corpus. Among them, 30K examples (15K I→O and 15K O→I) are used for the optional SFT step. The "+ I/O O/I RL" variant performs RL on all 60K examples (30K I→O and 30K O→I) without SFT. The "+ SFT + I/O O/I RL" variant uses the 30K split for SFT and runs RL on the remaining 30K examples (15K I→O and 15K O→I). The "+ SFT + white-box RL" variant shares the same 30K SFT split and performs RL on the remaining 30K examples (15K white-box I→O and 15K O→I).

E.2 Data-quality comparison setup

To isolate the impact of data quality, we compare our synthesized data against three representative datasets: PYX-Sub (Ding et al., 2024a), CodeI/O-Sub (Li et al., 2025), and Grounded-CoT (Jung et al., 2025), which correspond to the two paradigms in Section 1 (I/O CoT supervision vs. LLM-translated execution traces). For a fair comparison under the same budget, we sample matched-size training sets of 15K examples from each dataset and use a unified teacher, QwQ-32B (Team, 2024), to generate all CoT and trace translations with the same prompting. We then fine-tune the same Qwen2.5-Coder-Instruct model on the I→O prediction task using each 15K subset and report the results in Figure 3.

E.3 Ablation setups for data synthesis components

We conduct three SFT-only ablations, each using 15K training examples and the same fine-tuning protocol, evaluated on CRUXEval-O and LiveCodeBench-Exec. **Generating prompts with structural constraints:** *Full-constraint* follows Section 2.1 by prompting the generator with specified types/methods and explicit structural constraints, while *Weak-constraint* uses the same types/methods but removes structural constraints from the prompt. **Input synthesis:** using the same 15K code snippets, *Full-input* includes multiple input configurations (original inputs and type-aware mutations) and samples to 15K examples, whereas *Simple-input* keeps only one basic input per snippet. **Filtering by difficulty:** from the pool after execution filtering, *Filtered* applies difficulty-aware filtering before uniformly sampling 15K examples, while *No-filter* samples 15K examples directly without difficulty filtering.

α	CXEval-O	CXEval-I	LCB-O	Coverage	State	Path	Output	Avg
0.25	85.9	80.7	82.3	85.5	74.2	73.1	83.3	80.71
0.50	85.6	81.0	82.3	85.8	74.5	73.0	83.2	80.77
0.75	85.6	80.8	82.1	86.0	75.2	73.4	81.1	80.60

Table 8: Sensitivity analysis of the reward mixing coefficient α in Stage I. Higher is better for all metrics. Avg is the mean over the seven reported metrics, consistent with the main paper.

E.4 Construction of a Library-Involved I/O Prediction Benchmark

To evaluate transfer to library-dependent code, we construct a library-involved I/O prediction benchmark based on BigCodeBench. For each task, we extract an input–output pair from the example block in the task description (`complete_prompt`) and treat the task's (`canonical_solution`) as the executable reference program.

Concretely, we recover the *input* as the statements that define arguments and invoke the target function, and the *output* as the printed result shown in the example. At evaluation time, we provide the extracted input and ask the model to predict the stdout output produced by executing the canonical solution. We report exact match accuracy.

To ensure determinism and avoid environment-specific behavior, we filter out tasks involving randomness or external resources. This is done using keyword-level matching over both the prompt and the solution, targeting stochastic APIs (e.g., random) and file I/O (e.g., open, pathlib, pickle). The full blacklist is provided in Table 9. After filtering, we obtain 241 test cases for evaluation. A detailed example is shown in Figure 17.

F Qualitative Analysis: The Impact of Code Reasoning

F.1 Improved Fine-grained Code Execution Understanding

We conduct a qualitative analysis to demonstrate *ExecVerify*'s superior capability in tracing concrete execution steps compared to baselines. As illustrated in Figure 18, models trained solely with I/O O/I RL often struggle with control-flow logic, failing to evaluate guard conditions (e.g., if idx != idx2) and consequently "hallucinating" execution steps that do not occur. In contrast, *ExecVerify*, trained via White-box RL, correctly interprets conditional statements and skips invalid loop iterations. This confirms that the model faithfully tracks intermediate variable states and adheres to the strict program logic.

F.2 Benefit for Downstream Code Generation

We further investigate how the fine-grained execution reasoning capability transfers to downstream code generation tasks. Figure 19 presents a qualitative comparison on the problem "Count Substrings With K-Frequency Characters I," which requires identifying substrings where at least one character meets a frequency threshold. The baseline I/O O/I RL model generates syntactically correct code but contains a logical error, confusing the required condition ("at least one") with a universal one ("for all"). It incorrectly enforces a stricter constraint (if 0 < f < k: valid = False), rejecting valid substrings. In contrast, *ExecVerify*, enhanced by white-box reinforcement learning, correctly interprets the requirement and implements the logic using any(). This demonstrates that the fine-grained understanding of control flow and variable states acquired during reasoning training enables the model to perform a more accurate simulation of the program, leading to more robust handling of subtle logical constraints in code generation.

G Potential Risks

ExecVerify improves LLMs' ability to reason about and generate executable code. As with other code LLM advances, this capability could be misused to produce harmful scripts or assist vulnerability exploitation. Moreover, generated code may still be incorrect or insecure. Therefore, all outputs should be treated as assistive suggestions and validated via human review before deployment.

Category	Filtered libraries/APIs (keyword-level matching)
Randomness / stochasticity	random (shuffle, randint, choice(s), seed); numpy.random/np.random; torch.rand/torch.randn; secrets.
File I/O and filesystem traversal	open/with open, .read, .write; fileinput; pathlib; os.path; os ops (listdir, walk, scandir, remove, unlink, rmdir, mkdir, makedirs, rename, replace, stat, chmod, chown, getcwd, chdir); shutil; glob; tempfile.
Archives and compression	zipfile; tarfile; gzip; bz2; lzma.
Structured file readers/writers and external formats	csv (reader, dictreader, writer, dictwriter); pandas/pd (read_csv, read_table, read_excel, read_parquet, read_feather, read_json, read_pickle); writers (to_csv, to_excel); openpyxl.
Serialization / persistence	numpy/np (load, save, savez, savez_compressed); torch (save, load); pickle (load, dump); joblib (load, dump); json.load; yaml (load, safe_load).
Databases	sqlite3.connect.

Table 9: Blacklist used to exclude non-deterministic or environment-dependent BigCodeBench solutions when constructing the library-involved I/O prediction set.

An example of white-box questions

You are a programming expert. Your task is to analyze the Python code and answer the questions by simulating the execution step by step.

Here is the code content:

```
1   def test_rstrip(s):
2     result = s.rstrip()
3     for char in result:
4       if char.isalpha():
5         result = result.lstrip(char)
6       elif char.isdigit():
7         result = result.strip(char)
8       else:
9         result = result.rstrip(char)
10   return result
11   assert test_rstrip(" hello world ") == ????
```

Here are the questions:
Question1: Fill the assertion statement.
Question2: What is the value and type of the variable result after **Line 2** (result = s.rstrip()) is executed for the **1st time**?
Question3: Tracing the execution, which line is executed immediately after **Line 6** (elif char.isdigit():) is executed for the **1st time**?
Question4: Tracing the execution, which line is executed immediately after **Line 9** (result = result.rstrip(char)) is executed for the **1st time**?
Question5: What is the value and type of the variable result after **Line 5** (result = result.lstrip(char)) is executed for the **1st time**?
Question6: Tracing the execution, which line is executed immediately after **Line 4** (if char.isalpha():) is executed for the **3rd time**?

Guidelines for "next statement" questions:
- Determine the next line executed after the given statement.
- CRITICAL: Your answer MUST be the exact, verbatim source code of the next line -- copied character-for-character, including indentation.
- Do NOT include line numbers, quotes, backticks, comments, or any extra words.

Guidelines for "type & value" questions:
- STRICT FORMAT: value; type (Exactly one semicolon and one space).
- Example: 1; int 'hello'; str [1, 2]; list

ABSOLUTE FORMAT RULES (MUST FOLLOW):
- Output all answers one per line and in the listed order.
- For "next statement" answers: output ONLY the code statement string. Do not output line numbers!

Format your response strictly as follows:
<reasoning>
your step-by-step reasoning here
</reasoning>
<answer>
Answer for question1
...
Answer for question5
</answer>

Figure 15: A white-box question prompt used during reinforcement learning, combining a fully instrumented code snippet with multiple next-statement and value-and-type questions derived from its execution trace, together with strict formatting rules for the model's reasoning and answers.

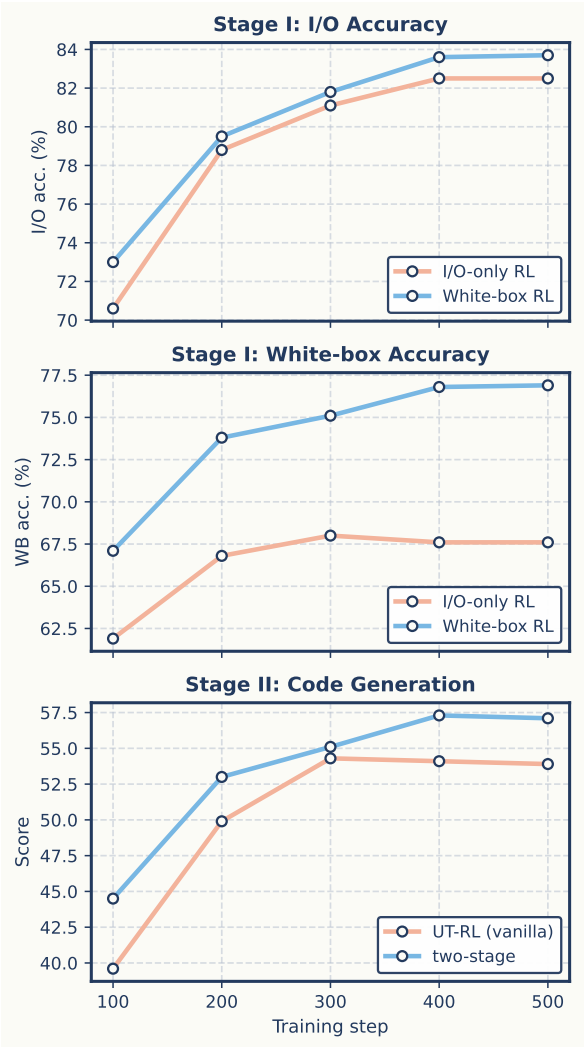

Figure 16: Stage I Reasoning and Stage II Code Generation: Training Curve Comparison.

An example of constructing a library-involved I/O prediction benchmark

Complete prompt (from complete_prompt):

```
from collections import Counter
import itertools
def task_func(d):
  """
  Count the occurrence of each integer in the values of the input dictionary,
  where each value is a list of integers, and return a dictionary with these counts.
  Requirements:
  - collections.Counter
  - itertools
  Example:
  >> d = {'a': [1, 2, 3, 1], 'b': [3, 4, 5], 'c': [1, 2]}
  >> count_dict = task_func(d)
  >> print(count_dict)
  {1: 3, 2: 2, 3: 2, 4: 1, 5: 1}
  """
```

Extracted I/O (from the example):

```
d = {'a': [1, 2, 3, 1], 'b': [3, 4, 5],
'c': [1, 2]}
count_dict = task_func(d)
print(count_dict)
stdout: {1: 3, 2: 2, 3: 2, 4: 1, 5:
1}
```

Canonical solution

```
from collections import Counter
import itertools
def task_func(d):
  count_dict = Counter(
    itertools.chain.from_iterable(d.values())
  )
  return dict(count_dict)
```

Final I/O prediction problem (model input):

```
Your task is to fill the assert statement.
from collections import Counter
import itertools
def task_func(d):
  count_dict = Counter(itertools.chain.from_iterable(d.values()))
  return dict(count_dict)
d = {'a': [1, 2, 3, 1], 'b': [3, 4, 5], 'c': [1, 2]}
count_dict = task_func(d)
assert count_dict == ????
```

Figure 17: An example of constructing the library-involved I/O prediction benchmark from BigCodeBench. We show the original `complete_prompt`, the extracted I/O pair, the executable `canonical_solution`, and the final I/O prediction problem.

Case Study: White-box RL vs. I/O O/I RL

Question.

```
1  from typing import List, Tuple
2
3  def find_closest_elements(numbers: List[float]) -> Tuple[float, float]:
4     closest_pair = None
5     distance = None
6     for idx, elem in enumerate(numbers):
7        for idx2, elem2 in enumerate(numbers):
8           if idx != idx2:
9              if distance is None:
10                distance = abs(elem - elem2)
11                closest_pair = tuple(sorted([elem, elem2])) # Line 11
12             else:
13                new_distance = abs(elem - elem2)
14                if new_distance < distance:
15                   distance = new_distance
16                   closest_pair = tuple(sorted([elem, elem2]))
17    return closest_pair
18
19 # assert find_closest_elements([1.0, 2.0, 3.9, 4.0, 5.0, 2.2]) == ????
```

Trace the execution of `find_closest_elements` with the given input. Is Line 11 (`closest_pair = ...`) executed? If yes, what is the value and type of `closest_pair` immediately after its **first execution**? If the line is unreachable, answer `Nil`.

I/O O/I RL (Incorrect)

Response. I trace the loops starting with `idx=0` (`elem=1.0`). Inside the inner loop, I start with `idx2=0` (`elem2=1.0`). At this point, `distance` is `None`, so the code executes Line 11: `closest_pair = tuple(sorted([1.0, 1.0]))`. Thus, the value is `(1.0, 1.0)` and the type is `tuple`.

Explanation. The model **fails to evaluate the guard condition** `if idx != idx2`. It erroneously enters the conditional block for the self-pair indices $(0, 0)$, hallucinating an execution of Line 11 that actually never happens in the program flow.

White-box RL (Correct)

Response. I start with `idx=0`. For the inner loop, the first iteration is `idx2=0`. Since `idx == idx2`, the condition `if idx != idx2` is false, so I skip the body. The loop continues to `idx2=1` (`elem2=2.0`). Now the condition holds. Since `distance` is `None`, Line 11 executes: `closest_pair = tuple(sorted([1.0, 2.0]))`. The value is `(1.0, 2.0)`.

Explanation. The model **correctly traces the control flow**. It recognizes that the condition `idx != idx2` prevents execution for the first pair $(0, 0)$, and correctly identifies the pair $(0, 1)$ as the first valid execution point.

Figure 18: Case study comparing execution tracing capabilities. The **I/O O/I RL** model overlooks the conditional statement (Line 8). In contrast, the **White-box RL** model strictly follows the program logic, correctly skipping the first iteration and identifying the true first assignment.

Case Study: Impact on Code Generation

Problem: Count Substrings With K-Frequency Characters I

Given a string `s` and an integer `k`, return the total number of substrings of `s` where **at least one character** appears **at least k times**.
Example: Input: `s = "abacb", k = 2`. Output: `4` ("aba", "abac", "abacb", "bacb").

I/O O/I RL (Incorrect)

```
class Solution:
   # Manually wrapped arguments to prevent ":" orphan
   def numberOfSubstrings(self, s: str,
                          k: int) -> int:
      n = len(s)
      count = 0

      for i in range(n):
         freq = [0] * 26
         for j in range(i, n):
            char_index = ord(s[j]) - ord('a')
            freq[char_index] += 1

            valid = True
            for f in freq:
               # INCORRECT LOGIC
               if 0 < f < k:
                  valid = False
                  break
            if valid:
               count += 1
      return count
```

Response Analysis. The model generates a full solution but fails at the core logic. It correctly sets up the loops and frequency counting, but the validation condition `if 0 < f < k: valid = False` enforces that *all* present characters must have frequency $\geq k$, violating the "at least one" requirement.

ExecVerify (Correct)

```
class Solution:
   # Manually wrapped arguments for clean layout
   def numberOfSubstrings(self, s: str,
                          k: int) -> int:
      n = len(s)
      count = 0

      for i in range(n):
         char_count = [0] * 26
         for j in range(i, n):
            char_count[ord(s[j]) - ord('a')] += 1
            # CORRECT LOGIC
            valid = any(c >= k for c in char_count)
            if valid:
               count += 1
      return count
```

Response Analysis. The model correctly identifies the condition. By tracing the execution state during training (White-box RL), it understands that the validation should pass as soon as *any* character meets the threshold, using `any(c >= k ...)` to correctly implement the logic.

Figure 19: Case study on code generation. The **I/O O/I RL** model produces syntactically correct code but fails logically by confusing "at least one" with "for all". In contrast, ***ExecVerify*** (Ours), trained via white-box reinforcement learning, correctly implements the semantic requirement using `any()`. This demonstrates that grounding the model in fine-grained runtime behavior equips it with a deeper understanding of program semantics, enabling it to accurately handle subtle logical constraints that surface-level I/O training often misses.