STEERABLE INSTRUCTION-FOLLOWING CODE SYNTHESIS VIA ACTOR-SCHEMA CO-EVOLUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Interpreting and following human instructions is a critical capability of large language models (LLMs) in automatic programming. However, synthesizing large-scale coding data paired with fine-grained, verifiable instructions remains largely unexplored, particularly when ensuring logical compatibility among multiple constraints. This setting reflects real-world user specifications, e.g., coding standards, dependency restrictions, and interface contracts. In this study, we propose IFCODEEVOLVE, an actor-schema co-evolution framework for instruction-following coding data generation. By representing each instruction as a parametric function schema, we construct a library that covers the vast constraint space via dynamic instantiation. Building upon this, a Monte Carlo Tree Search (MCTS) sampler is applied to efficiently navigate this space, utilizing actor model feedback as a dynamic termination signal. Furthermore, to progressively explore challenging problems, we introduce a co-evolving paradigm that iteratively advances both the actor model and the schema library based on sampler statistics. Empirical results demonstrate that IFCODEEVOLVE significantly boosts base model performance, with our 32B model achieving parity with proprietary SOTA models. Besides, we contribute IFCODEBENCH, a comprehensive human-verified benchmark equipped with solutions and robust AST-based verification.

1 INTRODUCTION

Large language models (LLMs) have recently achieved remarkable success in automatic programming, spanning domains from competitive programming (Li et al., 2022; Achiam et al., 2023; Seed et al., 2025) to software engineering (Jimenez et al., 2023; Zan et al., 2025; Guo et al., 2024). Beyond fundamental coding proficiency, an important factor leading to these advancements is the instruction following (IF) capability (Ouyang et al., 2022). In real-world software development, producing code that runs is often easier than producing code that adheres to user specifications, e.g., project-specific coding standards, dependency restrictions, or interface contracts. IF therefore serves as a bridge between fine-grained human specifications and executable logic, as illustrated in Figure 1.

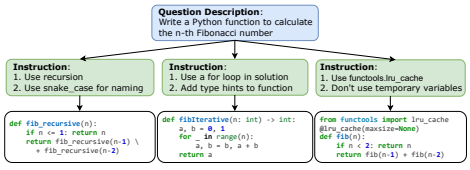

Figure 1: Instruction-driven code generation, where each instruction specifies a fine-grained, verifiable constraint on the implementation.

Improving the IF capability of LLMs necessitates large-scale coding data paired with high-quality instructions, i.e., problems augmented with verifiable constraints over the implementation. Such data currently depends heavily on manual curation. While recent approaches have explored automatic data synthesis using static brute-force sampling (Xu et al., 2024; Dong et al., 2024; Wei et al., 2023; Luo et al., 2023) or adversarial self-evolving frameworks (Zhao et al., 2025; Huang et al., 2025), they remain limited by two challenges. **(1) Validity and verification bottleneck**: directly prompting LLMs to generate multi-constraint coding problems struggles to guarantee logical consistency and solvability (Yan et al., 2025; Yang et al., 2025), and generating accurate verification logic (e.g., unit tests) is itself error-prone. **(2) Unstructured hard-sample exploration**: static frameworks rely primarily on the generator's intrinsic priors, resulting in repetitive data patterns and limited out-of-distribution coverage. Adversarial strategies, on the other hand, can easily hack the reward by producing unsolvable or ill-defined problems without carefully engineered reward designs.

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.

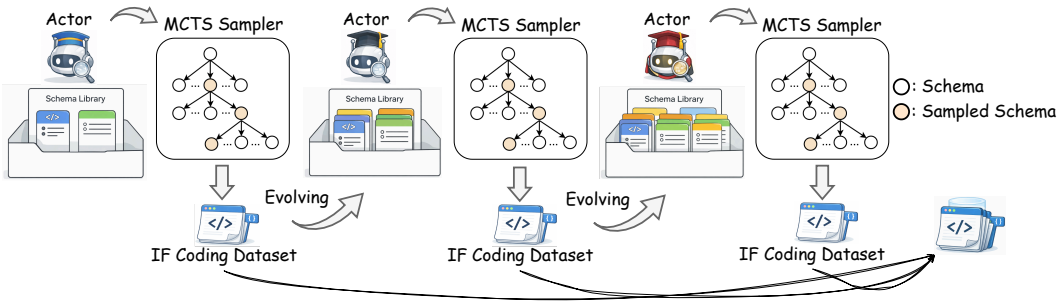

Figure 2: Illustration of actor-schema co-evolution paradigm for IF coding data generation.

Motivated by this, we propose IFCODEEVOLVE, a framework that optimizes logical validity and problem complexity by unifying progressive proof-by-construction and adversarial actor-schema co-evolution. We apply parametric instructions, which represent a vast instruction space within a compact library of schemas. This formulation transforms data synthesis into a steerable and sequential sampling process. Specifically, we frame synthesis as MCTS trajectory sampling integrated with a proof-by-construction validation strategy: at each step, the sampler navigates the combinatorial schema space to incrementally impose constraints, which are immediately instantiated and verified via code adaptation to ensure strict solvability.

Furthermore, this structured representation facilitates the exploration of high-complexity problems through actor-schema co-evolution: as the actor iteratively improves via post-training, the schema library autonomously evolves by merging synergistic primitives and mutating under-performing constraints. Unlike methods that improve data distribution by explicitly training the generator, we enhance data synthesis in a steerable manner, explicitly optimizing structural primitives to continuously challenge the evolving model. The overall paradigm is illustrated in Figure 2, where the sampler constructs IF coding data with iteratively improved actor and instruction schema.

We evaluate IFCODEEVOLVE on two public benchmarks across five LLMs with varying parameter sizes (1.3B to 32B). Empirical results demonstrate that our framework drives consistent performance gains across evolving procedures, and the larger models trained on our synthesized data achieve performance comparable to proprietary SOTA reasoning models. To complement these general baselines with fine-grained assessment, we introduce IFCODEBENCH, a curated benchmark equipped with executable tests and AST-based verification protocols. Finally, we posit that this paradigm, which extends the scope of evolution from the target model to the core components of the synthesis framework itself, offers a generalizable methodology for constructing self-adapting curricula in other domains, such as mathematical problem solving (Shao et al., 2024; Xin et al., 2025) and scientific discovery (Jumper et al., 2021; Du et al., 2025).

2 RELATED WORK

Instruction Tuning Instruction-following (IF) is a foundational capability of LLMs, requiring the model to interpret complex instructions and generate valid responses within an open-ended search space (Lou et al., 2023). To address the scarcity of high-quality instruction data, prior studies have resorted to manual annotation (Ouyang et al., 2022; Chung et al., 2024) or leveraged LLMs for large-scale synthesis from seed datasets (Wang et al., 2023; Xu et al., 2024; Dong et al., 2024). In the coding domain, methods such as OSS-Instruct (Wei et al., 2023) and Code Evol-Instruct (Luo et al., 2023) have proven effective at scaling up coding instruction data, but they target general code generation, where a query is paired with a functionally correct solution. In parallel, recent benchmarks (Zhou et al., 2023; Pyatkin et al., 2025; Jiang et al., 2024; Deshpande et al., 2025; Yan et al., 2025; Yang et al., 2025) have shifted attention toward constraint-conditioned code generation, where the solutions must simultaneously satisfy explicit user specifications, e.g., style, dependencies, and interface design.

Our work differs from prior coding data synthesis studies in both task and methodology: In task, rather than generating free-form coding queries, we synthesize problems jointly with verifiable instruction constraints. In methodology, prior approaches are largely self-instruct-based and thus inherit the generator's intrinsic bias. IFCODEEVOLVE instead frames synthesis as schema sampling,

proof-by-construction validation, and adversarial schema evolution, i.e., a form of harness engineering that makes task generation explicitly controllable, verifiable, and progressively challenging.

Self-Evolving Agents As LLMs demonstrate increasingly strong reasoning capabilities, a promising line of research investigates their potential to self-optimize agentic systems (Gao et al., 2025). Existing studies generally fall into four categories in terms of applications: **(1) prompt tuning**, where LLMs function as meta-optimizers to iteratively rewrite instructions and in-context exemplars, reducing reliance on manual configurations (Yang et al., 2023; Yuksekgonul et al., 2025); **(2) workflow construction**, which involves representing agentic topologies as computational graphs or code and optimizing interaction structures to solve complex problems (Hu et al., 2024; Zhang et al., 2024a); **(3) solution refinement**, in which the system engages in recursive feedback loops to critique, debug, and iteratively improve its own outputs (Madaan et al., 2023; Lin et al., 2025; Novikov et al., 2025); and **(4) data augmentation/generation**, where the models distill knowledge into datasets by synthesizing high-quality reasoning trajectories for self-training (Jumper et al., 2021; Silver et al., 2016; Zelikman et al., 2024; Xin et al., 2024), or employ adversarial strategies to generate challenging training instances (Zhao et al., 2025; Huang et al., 2025). Distinct from these prior approaches, IFCODEEVOLVE implements a meta-level evolution of the data generation mechanism itself. By iteratively evolving the instruction schema and actor, we sustain an adaptive curriculum for coding data generation.

3 METHOD

In this section, we introduce IFCODEEVOLVE, a co-evolving instruction-following coding data generation framework. We first formulate the problem and objective in Section 3.1, and elaborate the core components in Section 3.2. The implementation of the proposed co-evolving data synthesis framework is presented in Section 3.3 and Section 3.4.

3.1 PROBLEM FORMULATION

Given a set of coding problems, solutions, and functional unit tests $\mathcal{D} = \{(p_k, s_k, f^k_{\text{test}}(\cdot))\}_{k=1}^{N}$, where each coding problem p_k includes a question description, function signature, and input/output example, the goal of our proposed framework is to augment each instance with a set of instruction constraints:

$$p'_k, s'_k, \boldsymbol{\iota}_k = f_{\text{IFCodeEvolve}}(p_k, s_k) \tag{1}$$

where $\boldsymbol{\iota}_k = \{\iota_{k,j}\}_{j=1}^{M_k}$ is the set of instructions for the k-th problem. We focus on fine-grained, verifiable instructions, i.e., constraints that can be automatically validated via checking functions (e.g., naming conventions, data structure usage, or interface design), and impose them without altering the core algorithmic semantics, forcing the model to reason within a narrowed solution space.

3.2 INSTRUCTION REPRESENTATION AND SAMPLING

Parametric Instruction Previous methods (Dong et al., 2024; Xu et al., 2024) prompt LLMs to brainstorm instruction candidates using static text representations. Such representations treat parametric variations of the same logic (e.g., variable length > 5 vs. < 5) as distinct entities, leading to a redundant search space. In light of this, we apply parametric instruction (Yan et al., 2025), representing each instruction as an instantiable function schema, with configurable constraint attributes serving as parameters. Formally, a concrete instruction ι is derived by instantiating a schema τ which consists of:

- A textual skeleton with variable slots, e.g., "The length of variable names {`comparison`} exceed {`length`} characters.".
- An argument space defining valid types and scope for the slots, e.g., `comparison` $\in$ ["must", "must not"] and `length` $\in$ int.
- $f_{\text{IF-test}}(\cdot)$: An AST[1]-based verification function that maps a given solution to $\{0, 1\}$, indicating whether it strictly adheres to the instantiated instruction.

The instantiation process is driven by an LLM conditioned on the problem context, as detailed in Section 3.3. We construct an initial library $\mathcal{T}$ of 27 diverse schemas (see Appendix C), which serves as the foundation for the subsequent sampling procedure.

[1] https://docs.python.org/3/library/ast.html

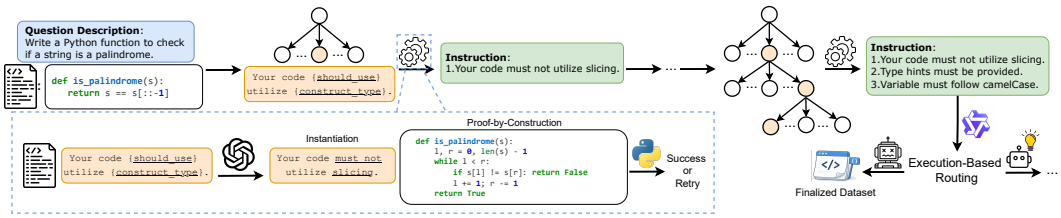

Figure 3: Illustration of multi-round IF coding data augmentation.

Monte Carlo Tree Search-based Sampler Building upon $\mathcal{T}$, we formulate IF data synthesis as a sequential tree search, where each node expansion instantiates a schema and a root-to-leaf trajectory constitutes the instruction set. To navigate this combinatorial space, we employ a Monte Carlo Tree Search (MCTS) (Guo et al., 2025) sampler that prioritizes schemas more likely to yield a challenging instruction set. Formally, let $\boldsymbol{\tau}_{<t} = \{\tau_1, \cdots, \tau_t\}$ denote the schemas sampled so far at step t, each corresponding to an instantiated instruction in $\boldsymbol{\iota}_{<t} = \{\iota_1, \cdots, \iota_t\}$. At each step, MCTS draws a candidate batch from the remaining library $\mathcal{T} \setminus \boldsymbol{\tau}_{<t}$ via softmax over a value score $Q(\boldsymbol{\tau}_{<t}, \tau)$:

$$\{\tau\}^C \sim \underset{\tau \in \mathcal{T} \setminus \boldsymbol{\tau}_{<t}}{\text{softmax}}\, Q(\boldsymbol{\tau}_{<t}, \tau) \tag{2}$$

Q combines an exploitation term, i.e., the empirical success rate of τ at the current state, smoothed by its global prior $P(\tau)$ (Zhai & Lafferty, 2017) with a UCB-style exploration term that favors under-visited schemas:

$$Q(\boldsymbol{\tau}_{<t}, \tau) = \underbrace{\frac{W(\boldsymbol{\tau}_{<t}, \tau) + \alpha \cdot P(\tau)}{N(\boldsymbol{\tau}_{<t}, \tau) + \alpha}}_{\text{exploitation}} + \underbrace{U(\boldsymbol{\tau}_{<t}, \tau)}_{\text{exploration}} \tag{3}$$

where W and N are the accumulated success and visit counts, α is a smoothing factor mitigating variance in low-visit nodes, and U follows the standard UCB form $\eta\sqrt{\ln N(\boldsymbol{\tau}_{<t})/N(\boldsymbol{\tau}_{<t}, \tau)}$ with weight η. For an unvisited schema, Q reduces to $P(\tau) + \eta$, so cold-start sampling falls back on globally strong schemas.

Discussion The utilization of parametric instructions enables us to cover a vast instruction space with a manageable set of schemas. As such, it allows the application of MCTS, which is grounded in statistical priors, to significantly enhance data synthesis efficiency.

3.3 Multi-Round Augmentation

As illustrated in Figure 3, given a coding problem and the schema library, our framework iteratively adds one sampled instruction from MCTS to the problem, validating solvability via a proof-by-construction instantiation and filtering instances through execution-based routing.

Proof-by-Construction Instantiation Let $\mathcal{D}_t$ denote the dataset at step t, where each instance $(p_t, s_t, \boldsymbol{\iota}_{<t})$ contains the problem, solution, and accumulated instructions. At each step, MCTS samples a batch of candidate schemas via Eq. 2, and an LLM generator π_{gen} produces the next state:

$$p_{t+1}, s_{t+1}, \boldsymbol{\iota}_{<t+1} = \pi_{\text{gen}}(p_t, s_t, \boldsymbol{\iota}_{<t}, \{\tau\}^C) \tag{4}$$

Specifically, π_{gen} is instructed to select the most compatible schema from the candidates and instantiate it to maximize difficulty. To guarantee validity, it must also produce a witness solution by adapting s_t to satisfy the new instruction, and the instantiation is accepted only if this solution passes both the cumulative IF verifiers and the functional unit tests. We include three examples in Appendix B.

Execution-Based Feedback Routing We use an actor LLM to probe solvability and drive a feedback-based curriculum. At each augmented state, we draw N candidate solutions from the actor, $s \sim \pi_{\text{actor}}(\cdot|p_t, \boldsymbol{\iota}_{<t})$, and treat the problem as *difficult* when both the functional pass rate and the all-instruction pass rate fall below a threshold λ:

$$\left(\mathbb{E}_s\left[f_{\text{test}}(s)\right] < \lambda\right) \wedge \left(\mathbb{E}_s\left[\prod_{\iota} f^{\iota}_{\text{IF-test}}(s)\right] < \lambda\right), \tag{5}$$

where the expectations are estimated via Best-of-N sampling (Wang et al., 2022). This indicator drives both routing and search. For routing: after augmentation at step t, instances the actor fails to solve are finalized, while solvable ones are propagated to the next iteration for further augmentation. For MCTS: a failure is counted as a win for the newly added schema, i.e., $W(\boldsymbol{\tau}_{<t}, \tau) \leftarrow W(\boldsymbol{\tau}_{<t}, \tau) + 1$.

3.4 Co-Evolving Paradigm

The augmentation paradigm above establishes a robust pipeline for IF code data generation, but a static schema library and a fixed actor cap problem diversity and difficulty at the actor's current capability. We therefore extend it into a self-evolving framework that co-evolves the actor and the schema library, allowing the synthesis frontier to scale with the model.

Actor Evolution The actor serves as a dynamic filter, routing instances it fails to solve into the finalized set. To produce progressively challenging problems, we post-train the actor on the newly synthesized data after each augmentation round; the upgraded actor then drives the next round, raising the difficulty bar. This induces a curriculum: patterns that the actor fails are absorbed through training, pushing the pipeline toward strictly harder instances that the improved actor still cannot solve.

Schema Evolution A fixed schema library lets the actor adapt to recurring patterns. We therefore refine $\mathcal{T}$ via two strategies: *composition* and *mutation*.

Composition leverages MCTS trajectory statistics to identify synergistic schema pairs. For each ordered pair (τ_i, τ_j), we compute the empirical success rate of τ_j conditioned on τ_i being the most recently sampled schema, aggregated across all visited states. A pair is merged into a composite schema $\tau_{i \oplus j}$ when this conditional rate exceeds both individual priors $P(\tau_i)$ and $P(\tau_j)$ by at least a margin δ, indicating that the two together yield greater difficulty than either alone.

Mutation targets the under-performing subset $\mathcal{T}_{\text{weak}} \subset \mathcal{T}$ with the lowest $P(\tau)$. For each $\tau \in \mathcal{T}_{\text{weak}}$, we retrieve instances where the actor passes both functional and IF tests, and prompt an LLM to (i) diagnose why τ is easily satisfied (e.g., loose parameter boundaries), (ii) propose a tightened schema τ' under which the actor's original solution would fail, and (iii) generate two positive witnesses to verify τ' remains logically sound. Only mutations passing validation are added to $\mathcal{T}$. This adversarial loop forces the actor to generalize rather than exploit fixed instruction patterns.

Computational Cost The computational cost of our pipeline primarily arises from (1) generator inference, (2) actor inference, and (3) actor training. MCTS itself adds negligible overhead, as each step involves only lightweight bookkeeping (e.g., dictionary lookups) without additional forward passes. Using GPT-OSS-120B (generator, reasoning effort = low) and Seed-Coder-8B (actor) on a single 8×A800 machine, the end-to-end cost across 3 rounds totals ∼15.99 GPU-hours (10.77h generator inference, 0.20h actor inference, 5.01h actor training). Generator inference thus dominates (67%), a cost shared by all LLM-based data synthesis methods. The cost also scales favorably: a larger schema library enlarges the MCTS candidate space but not the number of LLM calls per augmentation step, since the candidate size C and the mutation budget are fixed.

4 Experiment

4.1 Results of IFCodeEvolve

Source Dataset We clean up and construct a coding dataset by integrating three diverse public datasets, including MBPP (Austin et al., 2021), LeetCode (Easy Level) (Xia et al., 2025), ClassEval (Du et al., 2023). These datasets span distinct programming domains, covering basic logic, competitive algorithms, and class-level design. The final dataset comprises 1,565 coding problems, with 958, 519, and 88 samples derived from each respective source.

Benchmark We evaluate our method using two recent benchmarks dedicated to IF coding tasks: IFEvalCode (Yang et al., 2025) and CodeIF (Yan et al., 2025). We conduct evaluations on the

[2]GLM-5.2 frequently exceeds the maximum token limit on CodeIF, leading to truncated outputs and depressed scores; the standard generation budget is kept uniform across all models (16,000 tokens).

Table 1: Comparison on public IF-Coding benchmarks.

	IFEvalCode		CodeIF	
	Inst.	Prompt	Inst.	Prompt
Proprietary Models				
GPT-5.2	0.9391	0.8571	0.9460	0.6667
Gemini-3 Pro	0.9271	0.8286	0.8757	0.4023
Claude-4.5-Sonnet	0.9273	0.8048	0.8724	0.3879
Claude-4.5-Opus	0.9249	0.7952	0.9281	0.5575
Seed-1.6-thinking	0.9023	0.7238	0.7949	0.2040
Seed-1.8-thinking	0.9152	0.7857	0.6764	0.3621
Seed-2.0-thinking	0.9071	0.7952	0.9158	0.5489
Open-Source Models				
GLM-4.7	0.8989	0.7000	0.7163	0.3563
GLM-5.2[2]	0.8505	0.7619	0.1938	0.0517
Kimi-K2	0.8606	0.5714	0.8290	0.2931
GPT-OSS-20B	0.8894	0.7190	0.7090	0.3649
GPT-OSS-120B	0.9144	0.7571	0.5910	0.3362
Seed-OSS-36B	0.7363	0.6381	0.8231	0.3362
Qwen3-Coder-480B	0.8466	0.5762	0.8144	0.2358
DeepSeek-V3.2	0.8923	0.6741	0.8747	0.3506
DeepSeek-Coder-V2	0.7681	0.4190	0.7274	0.1494
Coding Data Synthesis Methods				
Qwen2.5-Coder-7B	0.6837	0.2800	0.5986	0.0718
+ OSS-Instruct	0.7188	0.4970	0.7216	0.1422
+ Code Evol-Instruct	0.7458	0.5274	0.7012	0.1503
+ IFCODEEVOLVE	0.8701	0.6143	0.8435	0.3007
Seed-Coder-8B	0.8051	0.4667	0.7337	0.1523
+ OSS-Instruct	0.8123	0.5001	0.7653	0.1469
+ Code Evol-Instruct	0.8253	0.5108	0.7890	0.1855
+ IFCODEEVOLVE	0.8519	0.6095	0.8520	0.3333

Table 2: Accuracy on IFEvalCode.

Actor	Original	+ 1st	+ 2nd	+ 3rd
Qwen2.5-Coder-7B	0.1523	0.2230	0.2523	0.2571
Seed-Coder-8B	0.2904	0.2952	0.3285	0.3142

Table 3: Performance after training on synthetic data.

	IFEvalCode		CodeIF	
	Inst.	Prompt	Inst.	Prompt
DeepSeek-Coder-1.3B	0.6446	0.2381	0.5725	0.0805
w/ RL Training	0.7822	0.4333	0.7985	0.2011
Qwen2.5-Coder-14B	0.7549	0.4190	0.7571	0.1351
w/ RL Training	0.8706	0.6571	0.8762	0.3563
Qwen2.5-Coder-32B	0.8492	0.5762	0.8161	0.2500
w/ RL Training	0.8962	0.7095	0.8883	0.4147

Table 4: Contamination analysis.

	IFEvalCode		CodeIF	
	$n = 5$	$n = 13$	$n = 5$	$n = 13$
Qwen2.5-Coder-7B	2.14%	0%	6.13%	0%
Seed-Coder-8B	5.06%	0%	3.02%	0%

Table 5: Ablation Study on Generator.

	IFEvalCode		CodeIF	
	Inst.	Prompt	Inst.	Prompt
Qwen2.5-Coder-7B	0.6837	0.2800	0.5986	0.0718
w/ Seed-1.6	0.8701	0.6143	0.8435	0.3007
w/ GPT-OSS-120B	0.8703	0.6238	0.8408	0.3161
w/ Qwen2.5-72B	0.7925	0.4857	0.7753	0.1667
Seed-Coder-8B	0.8051	0.4667	0.7337	0.1523
w/ Seed-1.6	0.8519	0.6095	0.8520	0.3333
w/ GPT-OSS-120B	0.8619	0.6333	0.8400	0.3100
w/ Qwen2.5-72B	0.8438	0.5857	0.8024	0.2098

Python subset (results for Java are provided in Appendix D). These benchmarks are curated from sources distinct from our seed dataset, ensuring no data overlap, and span diverse domains and task types, from algorithmic problems and data manipulation to code refactoring and debugging. As neither benchmark provides AST-based verifiers, and the rule-based checkers of IFEvalCode rely on string matching that can misjudge valid solutions, we validate the reliability of this judge against the deterministic AST-based verifiers of IFCODEBENCH (Section 4.3). We report metrics as defined in Zhou et al. (2023):

- *Inst.:* The average proportion of satisfied instructions per problem (Instruction-level pass rate).
- *Prompt:* The proportion of problems where *all* instructions are satisfied (Prompt-level pass rate).

Baselines We compare IFCODEEVOLVE against two representative coding data synthesis methods, OSS-Instruct (Wei et al., 2023) and Code Evol-Instruct (Luo et al., 2023), both of which target coding query generation. For fairness, we adapt both baselines to our experimental setting by keeping their original query-generation prompting strategies and matching the seed datasets, generator and actor models, and inference budget defined by the total number of generator and actor calls.

Base Model We evaluate the performance of 16 representative LLMs, comprising 7 proprietary models and 9 open-source models. Specifically, we employ `Seed-1.6` as the generator and utilize two coder models, i.e., `Qwen2.5-Coder-7B` and `Seed-Coder-8B`, as actor models, reporting their performance throughout the evolution. By default, we use the `Instruct` variants for all models.

Hyperparameter & Implementation We set $C = 3$, $\alpha = 10$, $\eta = 0.5$, $\lambda = 1$, and $\delta = 0.1$, and pick the 3 schemas with the lowest $P(\tau)$ for mutation. We apply GRPO (Shao et al., 2024) to post-train the actor across rounds. The reward combines functional correctness $r_{\text{func}} = \mathbb{I}(f_{\text{test}}(s))$ and instruction adherence rate $r_{\text{IF-dense}} = \frac{1}{|\iota|}\sum_{\iota} \mathbb{I}(f^{\iota}_{\text{IF-test}}(s))$, with a bonus for satisfying both:

$$r = r_{\text{func}} + r_{\text{IF-dense}} + 2 \cdot (r_{\text{func}} \cdot r_{\text{IF-dense}}). \tag{6}$$

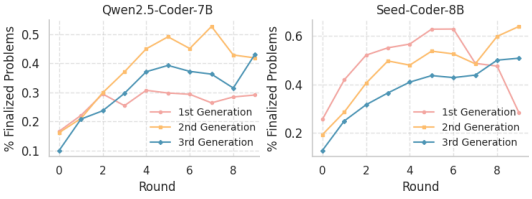

Figure 4: Percentage of finalized problems across rounds.

Table 6: Ablation Study on Sampler.

	IFEvalCode		CodeIF	
	Inst.	Prompt	Inst.	Prompt
`Qwen2.5-Coder-7B`	0.6837	0.2800	0.5986	0.0718
w/ BoN	0.8348	0.5571	0.8138	0.2299
w/ `Reject Sampling`	0.8401	0.5689	0.8199	0.2287
w/ `MCTS`	0.8703	0.6238	0.8408	0.3161
`Seed-Coder-8B`	0.8051	0.4667	0.7337	0.1523
w/ BoN	0.8405	0.5805	0.8415	0.2787
w/ `Reject Sampling`	0.8455	0.5887	0.8367	0.2689
w/ `MCTS`	0.8619	0.6333	0.8400	0.3100

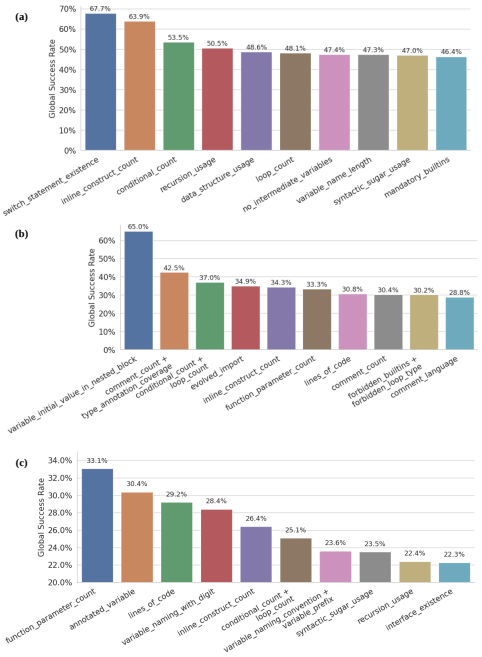

Figure 5: Top schema by global success rate. (a-c) correspond to Rounds 1–3.

Benchmarking Results As detailed in Table 1, proprietary models generally outperform open-source baselines, while the `GPT-OSS` series exhibits competitive capability. The metrics *Prompt* on CodeIF are significantly lower than those on IFEvalCode across all evaluated models. We attribute this gap to the higher number of instructions per problem in CodeIF, which imposes a substantially more rigorous challenge. Crucially, IFCODEEVOLVE consistently enhances both actor models, ultimately achieving performance comparable to SOTA reasoning models.

Table 2 reports the percentage of problems satisfying both unit tests and all instruction constraints. While both models show clear gains driven by enhanced instruction adherence, `Qwen2.5-Coder-7B` exhibits more improvement, as the stronger baseline of `Seed-Coder-8B` leaves limited room for further growth.

In Table 4, we evaluate the contamination of the training dataset generated across all iterations on IFEvalCode and CodeIF, and present the n-gram overlap results. It can be found that the contamination rate remains at 0% under the strict $n = 13$ standard, confirming that our generated data is free from leakage and that the observed performance gains stem from genuine capability enhancement. A TF-IDF-based check (Table 13) leads to the same conclusion.

Cross-scale Transferability To investigate the generalizability of our synthesized data, we employ the corpus generated with `Seed-Coder-8B` as the actor to train models of other scales and families. Specifically, we aggregate the synthesized data across all three iterations to post-train three models of varying sizes, utilizing Eq. 6 as the reward signal. The finalized dataset includes 4,241 coding problems, and more details can be found in Appendix C.2. As presented in Table 3, all models exhibit significant improvements in instruction following. Most notably, `Qwen2.5-Coder-32B` achieves performance parity with proprietary SOTA models, showing that the synthesized data transfers across model scales and families rather than being specific to the synthesis actor.

4.2 FURTHER ANALYSIS

Sampling Efficiency To analyze sampling efficiency, we track the ratio of finalized problems to total sampled candidates per round. Specifically, the actor's pass rate serves as a routing signal, where a problem is deemed finalized if the actor fails to solve the augmented problem (Section 3.3). An efficient sampler is expected to learn from interaction, progressively shifting the sampling

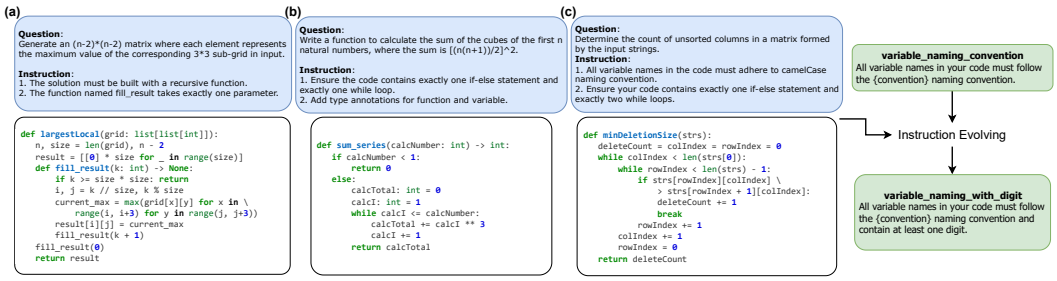

Figure 6: Representative examples of the three schema categories with adapted solutions. **(a)** A constraint from the initial library, i.e., `function_parameter_count` (2nd instruction). **(b)** A composed schema combining control flow constraints, i.e., `conditional_count + loop_count` (1st instruction). **(c)** A mutated schema illustrating the evolution from a standard convention (`variable_naming_convention`) to a stricter variant requiring digits (`variable_naming_with_digit`).

distribution towards hard-to-solve cases. As visualized in Figure 4, the finalization rate initially climbs and converges as instructions accumulate, demonstrating search efficiency. Besides, the two actor models exhibit distinct learning dynamics. For the stronger `Seed-Coder-8B`, the sampling process becomes increasingly challenging after evolution.

Diversity Analysis To examine whether the iterative pipeline collapses onto a narrow set of high-success-rate schemas, we track the top-10 schemas ranked by $P(\tau)$ across three rounds of `Seed-Coder-8B` (Fig. 5). First, the dominant schema changes every round, shifting from `switch_statement_existence` to `variable_initial_value_in_nested_block` and then `function_parameter_count`, with the top-10 largely refreshed round by round. Second, the top-1 success rate drops sharply ($67.7\% \rightarrow 65.0\% \rightarrow 33.1\%$), as previously effective schemas lose ground once the actor is trained on them, pushing the sampler toward harder configurations. Third, the makeup of top schemas evolves accordingly: Round 1 is dominated by atomic schemas, Round 2 introduces composed ones (e.g., `comment_count + type_annotation_coverage`), and Round 3 surfaces mutated variants (e.g., `variable_naming_with_digit`), mirroring the schema evolution in Section 3.4.

Ablation Study on π_{gen} We analyze the effect of the generator backbone π_{gen}. In addition to our default `Seed-1.6`, we test two open-source alternatives, i.e., `GPT-OSS-120B` and `Qwen2.5-72B`. Table 5 demonstrates that our method drives consistent improvements across all generator configurations. Importantly, the open-source `GPT-OSS-120B` achieves parity with the `Seed-1.6` model, underscoring the feasibility and robustness of our approach.

Ablation Study on Sampler We compare MCTS against two greedy alternatives: rejection sampling and Best-of-N (BoN, N=3), using GPT-OSS-120B as the generator for both actors. Both baselines extend a trajectory one schema at a time: rejection sampling accepts the first schema that causes the actor to fail (at most 3 attempts), whereas BoN draws N candidates simultaneously and selects the one that fails the actor. For fairness, both are matched to IFCODEEVOLVE's realized inference budget per round (Table 6).

MCTS consistently outperforms both samplers under a matched budget, for two reasons. First, MCTS itself adds no extra LLM calls whereas BoN explicitly multiplies inference cost per step. Second, greedy baselines spend their local budget chasing immediate actor failure and converge prematurely to shorter, easier trajectories (worsening with larger N): for `Seed-Coder-8B`, the average instruction counts across rounds are $2.48/2.82/2.66$ for BoN vs. $3.03/3.72/4.16$ for MCTS.

Additional Analysis More experiments are provided in Appendix D, including results on Java, ablation studies on the core modules of IFCODEEVOLVE, comparisons across reward functions, evolution patterns of schemas across rounds, instruction diversity under different schema evolution strategies, generalization to held-out schemas, and a TF-IDF-based contamination analysis.

Table 7: Performance of off-the-shelf models on IFCODEBENCH, grouped by parameter size and stratified by instruction count. None of the listed models is trained with our pipeline.

	$1 \sim 3$ Instructions					$4 \sim 6$ Instructions					≥ 7 Instructions				
	Func.	Unit Test		LLM		Func.	Unit Test		LLM		Func.	Unit Test		LLM	
		Inst.	Prompt	Inst.	Prompt		Inst.	Prompt	Inst.	Prompt		Inst.	Prompt	Inst.	Prompt
Proprietary Models															
GPT-5.2	0.9583	0.9768	0.9652	0.9826	0.9513	0.9693	0.9800	0.9080	0.9720	0.8773	0.8720	0.9453	0.7680	0.9542	0.8160
Gemini-3 Pro	0.9166	0.9803	0.9652	0.9953	0.9861	0.8429	0.9775	0.9157	0.9767	0.9386	0.7680	0.9602	0.7920	0.9361	0.7840
Claude-4.5-Sonnet	0.9236	0.9534	0.9097	0.9710	0.9375	0.8507	0.9604	0.8352	0.9583	0.8390	0.7920	0.9618	0.7360	0.9650	0.7600
Claude-4.5-Opus	0.9375	0.9806	0.9444	0.9791	0.9375	0.9003	0.9653	0.8735	0.9650	0.9003	0.8160	0.9541	0.7920	0.9254	0.7760
Seed-1.6-thinking	0.9305	0.9324	0.8819	0.9675	0.9305	0.8199	0.9448	0.7892	0.9480	0.7624	0.6560	0.8840	0.6560	0.9337	0.6000
Seed-1.8-thinking	0.9027	0.9571	0.9236	0.9745	0.9305	0.8084	0.9495	0.8199	0.9515	0.7892	0.6800	0.9287	0.7120	0.9516	0.7280
Seed-2.0-thinking	0.9310	0.9830	0.9580	0.9910	0.9720	0.8870	0.9260	0.8850	0.9780	0.9000	0.7760	0.8710	0.7840	0.9740	0.8640
400B+ Open-Source Models															
GLM-5.2	0.9030	0.9620	0.9310	0.9510	0.9380	0.8310	0.9390	0.8580	0.9300	0.8700	0.7040	0.9150	0.7600	0.9070	0.7520
Kimi-K2	0.8125	0.8331	0.7291	0.8877	0.7361	0.7241	0.8684	0.5325	0.8637	0.5172	0.6880	0.8268	0.2080	0.8288	0.2240
DeepSeek-V3.2	0.8680	0.8412	0.7569	0.9050	0.7708	0.8084	0.9310	0.6973	0.9141	0.6360	0.7120	0.9186	0.4880	0.9057	0.4720
Qwen3-Coder-480B	0.8611	0.8033	0.6527	0.8761	0.6944	0.8199	0.8867	0.6015	0.8805	0.5593	0.7600	0.8830	0.3680	0.8622	0.3440
100B+ Open-Source Models															
GLM-4.7	0.8819	0.9569	0.9305	0.9363	0.9097	0.7164	0.8834	0.7739	0.8371	0.7279	0.6400	0.9133	0.7360	0.8267	0.6960
DeepSeek-Coder-V2	0.8194	0.6806	0.5000	0.7766	0.4861	0.6858	0.8109	0.3984	0.7763	0.3371	0.6160	0.7776	0.1280	0.7211	0.0960
GPT-OSS-120B	0.9236	0.9623	0.9375	0.9745	0.9444	0.8659	0.9399	0.8505	0.9295	0.8314	0.7440	0.8960	0.7040	0.8477	0.7040
20B+ Open-Source Models															
Seed-OSS-36B	0.8888	0.9108	0.8541	0.9594	0.8958	0.7816	0.9246	0.7739	0.9551	0.8314	0.6640	0.8839	0.6400	0.9660	0.7920
Qwen2.5-Coder-32B	0.6805	0.8038	0.6805	0.8750	0.7013	0.5977	0.8658	0.5440	0.8819	0.5747	0.5600	0.8320	0.2880	0.8537	0.3040
GPT-OSS-20B	0.9027	0.9187	0.8541	0.9375	0.8819	0.7624	0.8884	0.7432	0.8302	0.6973	0.6240	0.8301	0.5600	0.7210	0.5200
7B+ Open-Source Models															
DeepSeek-Coder-V2-Lite	0.7777	0.4490	0.2013	0.6145	0.2083	0.7279	0.5722	0.0766	0.5504	0.0498	0.6400	0.5996	0.0160	0.5520	0.0160
Qwen2.5-Coder-14B	0.7083	0.6127	0.4166	0.7407	0.4236	0.6628	0.7505	0.2643	0.7272	0.2528	0.5840	0.6874	0.0880	0.6611	0.0800
Seed-Coder-8B	0.7291	0.6664	0.4652	0.7581	0.4375	0.6283	0.7789	0.3678	0.7657	0.3295	0.5920	0.7360	0.1120	0.7370	0.1200
Qwen2.5-Coder-7B	0.6875	0.5025	0.2638	0.6469	0.2638	0.6743	0.6579	0.1417	0.6127	0.1111	0.5840	0.6425	0.0240	0.5956	0.0320

4.3 IFCODEBENCH

Positioning IFCODEBENCH is a community benchmark curated as a by-product of our synthesis pipeline, intended for future research on IF code generation rather than to evaluate IFCODEEVOLVE itself. All main results in Section 4.1 use the independent IFEvalCode and CodeIF benchmarks, which share no overlap with our training data (cf. Table 4). The models in Table 7 are off-the-shelf baselines, none trained with our pipeline.

Construction We curated IFCODEBENCH from our generated pool through a human-in-the-loop process based on a rubric of four binary sanity checks and two scalar quality metrics. An LLM-based filter first surfaced high-potential candidates (Value = 3); human annotators re-applied the same rubric, retaining only unanimously confirmed instances and yielding 530 problems (Appendix E).

Metrics IFCODEBENCH is partitioned into three subsets by instruction count, and we report functional correctness (*Func.*, pass rate on unit tests) along with *Inst.* and *Prompt* (defined as in Section 4.1), computed with both execution-based checkers and an LLM-based judge (Table 7).

Findings LLM-judge results align closely with execution-based ones (Pearson $r = 0.984$ across all 21 models), validating the reliability of our automatic verifiers. Across models, performance inversely correlates with instruction count, and a substantial gap separates proprietary models from smaller open-source ones, which struggle to satisfy multiple concurrent constraints.

5 CONCLUSION

We presented IFCODEEVOLVE, a co-evolutionary data synthesis framework that unifies parametric instruction design with MCTS-guided validation. By structurally evolving the schema library alongside the actor model, our approach guarantees strict solvability while continuously pushing the frontier of problem complexity. Empirical results demonstrate that IFCODEEVOLVE effectively overcomes the limitations of static data generation, establishing a robust foundation for training next-generation instruction-following code LLMs.

AI USE STATEMENT

In this work, we used generative AI tools to generate synthetic datasets, which is the core of the proposed method: the generator π_{gen} (`Seed-1.6`, `GPT-OSS-120B`, or `Qwen2.5-72B`, depending on the experiment) instantiates parametric schemas, produces witness solutions, and mutates schemas, and an LLM paraphrases the instructions of finalized problems. LLMs also pre-filter candidates when curating IFCODEBENCH and serve as judges in evaluation. All LLM roles, model identities, prompt templates, and hyperparameters are documented in Section 4.1 and Appendices B and F. We have not used generative AI tools to develop theoretical models or conceptual frameworks, propose or refine hypotheses, design or provide feedback on the methodology or experiments, implement methods, assist with translation, clean or reformat the seed datasets, or interpret results, and formulating or proving mathematical claims and qualitative or thematic data analysis are not applicable to this work. Additionally, we used generative AI tools to polish the writing of the paper and to make local edits to software code. We have reviewed all AI-assisted work: every synthesized instance is validated by execution against functional unit tests and AST-based verifiers, every instance of IFCODEBENCH was additionally verified by human annotators using the rubric in Appendix E.2, and all AI-polished text and AI-edited code were reviewed by the authors. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REFERENCES

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

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

Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. Scaling instruction-finetuned language models. *Journal of Machine Learning Research*, 25(70):1–53, 2024.

Kaustubh Deshpande, Ved Sirdeshmukh, Johannes Baptist Mols, Lifeng Jin, Ed-Yeremai Hernandez-Cardona, Dean Lee, Jeremy Kritz, Willow E Primack, Summer Yue, and Chen Xing. Multichallenge: A realistic multi-turn conversation evaluation benchmark challenging to frontier llms. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 18632–18702, 2025.

Guanting Dong, Keming Lu, Chengpeng Li, Tingyu Xia, Bowen Yu, Chang Zhou, and Jingren Zhou. Self-play with execution feedback: Improving instruction-following capabilities of large language models. *arXiv preprint arXiv:2406.13542*, 2024.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *arXiv preprint arXiv:2308.01861*, 2023.

Yuanqi Du, Botao Yu, Tianyu Liu, Tony Shen, Junwu Chen, Jan G Rittig, Kunyang Sun, Yikun Zhang, Zhangde Song, Bo Zhou, et al. Accelerating scientific discovery with autonomous goal-evolving agents. *arXiv preprint arXiv:2512.21782*, 2025.

Huan-ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Yiran Wu, et al. A survey of self-evolving agents: On path to artificial super intelligence. *arXiv preprint arXiv:2507.21046*, 2025.

Irving J Good. The population frequencies of species and the estimation of population parameters. *Biometrika*, 40(3-4):237–264, 1953.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

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

Mark O Hill. Diversity and evenness: a unifying notation and its consequences. *Ecology*, 54(2): 427–432, 1973.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*, 2024.

Chengsong Huang, Wenhao Yu, Xiaoyang Wang, Hongming Zhang, Zongxia Li, Ruosen Li, Jiaxin Huang, Haitao Mi, and Dong Yu. R-zero: Self-evolving reasoning llm from zero data. *arXiv preprint arXiv:2508.05004*, 2025.

Yuxin Jiang, Yufei Wang, Xingshan Zeng, Wanjun Zhong, Liangyou Li, Fei Mi, Lifeng Shang, Xin Jiang, Qun Liu, and Wei Wang. Followbench: A multi-level fine-grained constraints following benchmark for large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4667–4688, 2024.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. Highly accurate protein structure prediction with alphafold. *nature*, 596(7873):583–589, 2021.

Jia Li, Chongyang Tao, Jia Lio, Ge Li, Zhi Jin, Huangzhao Zhang, Zheng Fang, and Fang Liu. Large language model-aware in-context learning for code generation. *ACM Transactions on Software Engineering and Methodology*, 34(7):1–33, 2025.

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

Jiaye Lin, Yifu Guo, Yuzhen Han, Sen Hu, Ziyi Ni, Licheng Wang, Mingguang Chen, Hongzhang Liu, Ronghao Chen, Yangfan He, et al. Se-agent: Self-evolution trajectory optimization in multi-step reasoning with llm-based agents. *arXiv preprint arXiv:2508.02085*, 2025.

Renze Lou, Kai Zhang, and Wenpeng Yin. A comprehensive survey on instruction following. *arXiv preprint arXiv:2303.10475*, 1, 2023.

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

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36:46534–46594, 2023.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

Valentina Pyatkin, Saumya Malik, Victoria Graf, Hamish Ivison, Shengyi Huang, Pradeep Dasigi, Nathan Lambert, and Hannaneh Hajishirzi. Generalizing verifiable instruction following. *arXiv preprint arXiv:2507.02833*, 2025.

ByteDance Seed, Yuyu Zhang, Jing Su, Yifan Sun, Chenguang Xi, Xia Xiao, Shen Zheng, Anxiang Zhang, Kaibo Liu, Daoguang Zan, et al. Seed-coder: Let the code model curate data for itself. *arXiv preprint arXiv:2506.03524*, 2025.

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

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions. In *Proceedings of the 61st annual meeting of the association for computational linguistics (volume 1: long papers)*, pp. 13484–13508, 2023.

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

Yunhui Xia, Wei Shen, Yan Wang, Jason Klein Liu, Huifeng Sun, Siyue Wu, Jian Hu, and Xiaolong Xu. Leetcodedataset: A temporal dataset for robust evaluation and efficient training of code llms. *arXiv preprint arXiv:2504.14655*, 2025.

Huajian Xin, Daya Guo, Zhihong Shao, Zhizhou Ren, Qihao Zhu, Bo Liu, Chong Ruan, Wenda Li, and Xiaodan Liang. Deepseek-prover: Advancing theorem proving in llms through large-scale synthetic data. *arXiv preprint arXiv:2405.14333*, 2024.

Ran Xin, Chenguang Xi, Jie Yang, Feng Chen, Hang Wu, Xia Xiao, Yifan Sun, Shen Zheng, and Ming Ding. Bfs-prover: Scalable best-first tree search for llm-based automatic theorem proving. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 32588–32599, 2025.

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

Pengyu Xue, Linhao Wu, Zhen Yang, Chengyi Wang, Xiang Li, Yuxiang Zhang, Jia Li, Ruikai Jin, Yifei Pei, Zhaoyan Shen, et al. Classeval-t: Evaluating large language models in class-level code translation. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):1421–1444, 2025.

Kaiwen Yan, Hongcheng Guo, Xuanqing Shi, Jingyi Xu, Yaonan Gu, and Zhoujun Li. Codeif: Benchmarking the instruction-following capabilities of large language models for code generation. *arXiv preprint arXiv:2502.19166*, 2025.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*, 2023.

Jian Yang, Wei Zhang, Shukai Liu, Linzheng Chai, Yingshui Tan, Jiaheng Liu, Ge Zhang, Wangchunshu Zhou, Guanglin Niu, Zhoujun Li, et al. Ifevalcode: Controlled code generation. *arXiv preprint arXiv:2507.22462*, 2025.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Pan Lu, Zhi Huang, Carlos Guestrin, and James Zou. Optimizing generative ai by backpropagating language model feedback. *Nature*, 639(8055):609–616, 2025.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, et al. Multi-swe-bench: A multilingual benchmark for issue resolving. *arXiv preprint arXiv:2504.02605*, 2025.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D Goodman. Star: Self-taught reasoner bootstrapping reasoning with reasoning. In *Proc. the 36th International Conference on Neural Information Processing Systems*, volume 1126, 2024.

Chengxiang Zhai and John Lafferty. A study of smoothing methods for language models applied to ad hoc information retrieval. In *Acm sigir forum*, volume 51, pp. 268–276. ACM New York, NY, USA, 2017.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, et al. Aflow: Automating agentic workflow generation. *arXiv preprint arXiv:2410.10762*, 2024a.

Shudan Zhang, Hanlin Zhao, Xiao Liu, Qinkai Zheng, Zehan Qi, Xiaotao Gu, Xiaohan Zhang, Yuxiao Dong, and Jie Tang. Naturalcodebench: Examining coding performance mismatch on humaneval and natural user prompts. *arXiv preprint arXiv:2405.04520*, 2024b.

Andrew Zhao, Yiran Wu, Yang Yue, Tong Wu, Quentin Xu, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. Absolute zero: Reinforced self-play reasoning with zero data. *arXiv preprint arXiv:2505.03335*, 2025.

Jeffrey Zhou, Tianjian Lu, Swaroop Mishra, Siddhartha Brahma, Sujoy Basu, Yi Luan, Denny Zhou, and Le Hou. Instruction-following evaluation for large language models. *arXiv preprint arXiv:2311.07911*, 2023.

A LIMITATION

Scope of Constraints Our schema library targets constraints that can be verified by static analysis, such as naming conventions, control-flow structures, interface design, and library usage. Many real-world requirements are instead behavioral or repository-level, and fine-grained IF evaluation in such settings remains largely unexplored, as existing software engineering benchmarks mostly assess end-to-end task success rather than per-constraint compliance. IFCODEEVOLVE is not tied to a particular constraint family: new constraint types can be incorporated by adding the corresponding schemas and verifiers, as in our Java extension (Appendix D), and the framework can in principle be applied to bug fixing when reference patches and executable tests are available. We leave richer behavioral and debugging constraints, as well as larger schema libraries, to future work.

Controllability of Schema Evolution Composition and mutation refresh the schema library automatically based on MCTS statistics and adversarial analysis, so the direction of evolution cannot be specified in advance. This is, however, an interface-level limitation rather than an inherent lack of control: every schema, including evolved ones, is stored explicitly with its instruction template, parameters, and verifier. Practitioners can therefore exclude undesirable schemas through a blacklist, restrict MCTS sampling to selected schema categories, enforce a particular family through a whitelist, and review evolved schemas before they are added to the library. More controllable evolution mechanisms, e.g., constraint-aware mutation, are left to future work.

B IMPLEMENTATION

Pseudocode We provide the pseudocode of IFCODEEVOLVE in Algo. 1. The `Reset` means that, at the start of each round, we clear the MCTS statistics from the previous round and rebuild the sampler using the post-trained actor as the warm start. The same applies to schema-composition statistics, which are computed only from trajectories collected within the current round rather than aggregated across rounds. The rationale is that actor post-training substantially reshapes the hardness landscape, i.e., patterns that are high-yield against one actor may become trivial for the next, so carrying over stale statistics would bias the sampler. This effect is consistent with the round-over-round shifts in the top-10 schema distribution (Fig. 5).

In addition, we apply a dynamic expansion rule: if the candidate pool size drops below 500 (due to routing), each problem is augmented twice in the next iteration. This redundant augmentation ensures a consistent data supply. To enhance diversity, we employ an LLM to perform semantics-preserving paraphrasing for instructions of finalized questions. The prompt can be found in Appendix F.

Algorithm 1 IFCODEEVOLVE

Require: Seed Dataset $\mathcal{D}_{\text{seed}}$, Instruction Schema Library $\mathcal{T}$, Generator π_{gen}, Actor π_{actor}

Initialize sampler π_{MCTS} using $\mathcal{T}$
$\mathcal{D}_{\text{final}} \leftarrow \emptyset$
for iteration $k = 1$ to K **do**
  // Phase 1: Multi-Round Augmentation
  $\mathcal{D}_{\text{new}} \leftarrow \text{AugmentData}(\mathcal{D}_{\text{seed}}, \mathcal{T}, \pi_{\text{gen}}, \pi_{\text{actor}}, \pi_{\text{MCTS}})$
  $\mathcal{D}_{\text{final}} \leftarrow \mathcal{D}_{\text{final}} \cup \mathcal{D}_{\text{new}}$
  // Phase 2: Actor Evolving
  $\pi_{\text{actor}} \leftarrow \text{PostTrain}(\pi_{\text{actor}}, \mathcal{D}_{\text{new}})$
  // Phase 3: Sampler Evolving
  $\pi_{\text{MCTS}} \leftarrow \text{Reset}(\mathcal{D}_{\text{new}}, \pi_{\text{MCTS}}, \pi_{\text{actor}})$
  // Phase 4: Schema Evolving
  $\{\tau_{i\oplus j}\} \leftarrow \text{Composition}(\pi_{\text{MCTS}}, \pi_{\text{actor}})$
  $\mathcal{T} \leftarrow \mathcal{T} \cup \{\tau_{i\oplus j}\}$
  $\mathcal{T}_{\text{weak}} \leftarrow \text{SelectWeakSchema}(\mathcal{T}, \pi_{\text{MCTS}})$
  $\mathcal{T}'_{\text{weak}} \leftarrow \text{Mutation}(\mathcal{D}_{\text{final}}, \mathcal{T}_{\text{weak}}, \pi_{\text{gen}})$
  $\mathcal{T} \leftarrow \mathcal{T} / \mathcal{T}_{\text{weak}} \cup \mathcal{T}'_{\text{weak}}$
end for
return $\mathcal{D}_{\text{final}}, \pi_{\text{actor}}$

Running environment. The experiments are conducted on a single Linux server with Intel(R) Xeon(R) Platinum 8336C CPU, 1.9Ti RAM, and 8 NVIDIA A800-SXM4-80GB. Our method is implemented on PyTorch 2.8.0 and Python 3.11.2.

Training All the RL training are based on an open-source framework verl[3]. We set the group size to $G = 8$, with a global training batch size of 256 and a mini-batch size of 64. We employ dynamic batching with a maximum of 16,384 tokens per GPU. The learning rate is fixed at 1×10^{-6} with 10 warmup steps. We employ an asymmetric PPO clipping range of $[0.2, 0.28]$ and disable the KL penalty. All training is conducted with FSDP2 and vLLM integration, supporting a maximum sequence length of 2048 tokens.

Examples of Instruction Instantiation For the parametric instruction, we now present three examples of instantiation in Figure 7.

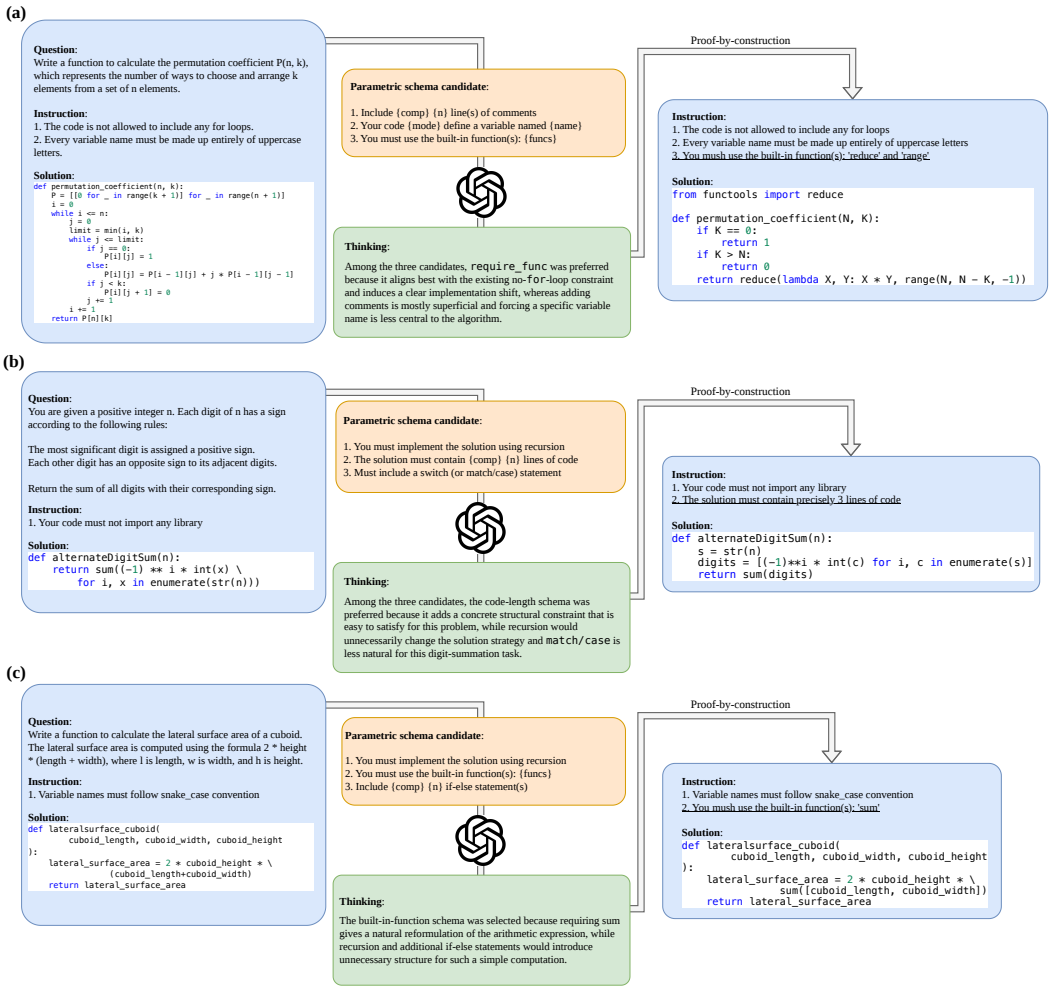

Figure 7: Case studies of parametric schema instantiation. The adapted solution is considered valid only if it passes both the functional unit tests and the corresponding AST-based verifier.

[3] https://github.com/verl-project/verl

C DATASET

C.1 SCHEMA

The library of parametric schema is detailed in Table 8. For improved presentation, certain schema identifiers have been abbreviated (e.g., `variable_existence` is denoted as `var_existence`).

Each schema is paired with an AST-based verification function, which dynamically adapts to the instantiated parameters. Taking the `variable_existence` (Code 1) and `conditional_count` (Code 2) as examples, the verifier traverses the Abstract Syntax Tree to identify all variable assignments and control flow structures.

In `variable_existence`, the parameter `mode` governs the validation logic: if set to 'should', the function confirms the presence of the specified variable; if 'should not', it ensures its absence. As for `conditional_count`, the parameter `comparison` (e.g., 'at least', 'exactly') combined with a target `count` defines the numerical constraint. The verifier counts the occurrences of `ast.If` and `ast.Compare` nodes, subsequently evaluating whether the total satisfies the specified relational condition.

Code 1 AST-based verification logic for `variable_existence`.

```python
import ast

def check_variable_existence(code, variable_name, mode):
    tree = ast.parse(code)
    assert mode in ["should", "should not"]
    defined_vars = set()
    for node in ast.walk(tree):
        if isinstance(node, ast.Assign):
            for target in node.targets:
                if isinstance(target, ast.Name):
                    defined_vars.add(target.id)
        # Also catch annotated assignment: x: int = 1
        elif isinstance(node, ast.AnnAssign):
            if isinstance(node.target, ast.Name):
                defined_vars.add(node.target.id)
    exists = variable_name in defined_vars
    if str(mode).lower() == "should not":
        return not exists
    return exists
```

Code 2 AST-based verification logic for `conditional_count`.

```python
import ast

def check_conditional_count(code, count, comparison):
    tree = ast.parse(code)
    assert comparison in ["at least", "at most", "exactly"]
    if_nodes = [n for n in ast.walk(tree) if isinstance(n, ast.If)]
    actual = len(if_nodes)
    target = int(count)
    if "at least" in comparison: return actual >= target
    if "at most" in comparison: return actual <= target
    if "exactly" in comparison: return actual == target
    return False
```

Extension to Java. To verify that our framework generalizes beyond Python, we adapt the schema library to Java. Most schemas transfer directly across languages, since the underlying logic (e.g., variable constraints, naming conventions, control-flow structures) is largely language-agnostic. We retain these schemas with minimal changes and adapt only the language-specific ones, e.g., `sugar_usage`

Table 8: Library of parametric instruction schema.

Schema	Instruction Template	Parameter
Category: Variable & Structures		
var_existence	Your code {mode} define a variable named {name}.	mode: [should, should not]; name: *str*
no_inter_var	Your code must not use intermediate variables.	-
naming_conv	Variable names must follow {style} convention.	style: [camelCase, PascalCase, snake_case]
global_var	Must define a global variable named {name}.	name: *str*
init_value	Define variable {name} initialized with {val}.	name: *str*; val: *str*
name_len	Variable name length {mode} exceed {n} characters.	mode: [should, should not]; n: *int*
var_prefix	Variable names must start with prefix '{pre}'.	pre: *str*
var_suffix	Variable names must end with suffix '{suf}'.	suf: *str*
data_struct	The data structure {ds} {mode} be used.	ds: [dict, set, list, tuple, ...]; mode: [must, must not]
Category: Logic & Control Flow		
loop_count	Include {comp} {n} {type} loop(s).	comp: [at least, at most, exactly]; type: [while, for]; n: *int*
forbid_loop	Your code must not use any {type} loops.	type: [while, for]
cond_count	Include {comp} {n} if-else statement(s).	comp: [exactly, at least, at most]; n: *int*
switch_stmt	Must include a switch (or match/case) statement.	-
recursion	You must implement the solution using recursion.	-
Category: Interface & Type		
interface	You should define a {type} named {name}.	type: [class, interface]; name: *str*
type_hint	Type annotations must be used for functions and vars.	-
func_def	Your code must define a function named {name}.	name: *str*
param_count	Function {name} must accept exactly {n} params.	name: *str*; n: *int*
Category: Library & Tools		
import_lib	Your code {mode} import the library {lib}.	mode: [must, must not]; lib: *str*
no_import	Your code must not import any library.	-
forbid_func	You are not allowed to use built-in(s): {funcs}.	funcs: *list[str]*
require_func	You must use the built-in function(s): {funcs}.	funcs: *list[str]*
sugar_usage	Your code {mode} utilize {type}.	mode: [must, must not]; type: [list comp, lambda, ternary operator, generator exp, ...]
sugar_count	Utilize {comp} {n} {type}.	type: [list comp, lambda, ternary operator, generator exp, ...]; n: *int*; comp: [at least, at most, exactly]
Category: Style & Formatting		
comment_cnt	Include {comp} {n} line(s) of comments.	comp: [exactly, more than, less than]; n: *int*
comment_lang	Comments must be written in {lang}.	lang: [en, zh]
code_lines	The solution must contain {comp} {n} lines of code.	comp: [exactly, at least, at most]; n: *int*

(which references Python-specific syntactic sugar) and class-related constraints (which we expand for Java given its object-oriented focus). For instruction verification, we replace the Python `ast` module with `tree-sitter`[4], which provides a language-agnostic concrete syntax tree from which we extract the same structural signals (e.g., variable assignments, conditional nodes) used by the original verifiers.

C.2 Training Dataset

In Section 4.1, we evaluate the efficacy of our approach by post-training a series of LLMs across different parameter scales using the synthesized dataset. The dataset was curated via IFCodeEvolve, employing `Seed-Coder-8B` as the base actor model. Our pipeline yielded a total of 4,241 high-quality IF coding instances, with an average of approximately 1,400 problems generated per iteration. The taxonomic distribution of schema categories and the corresponding instruction counts are detailed in Figure 8.

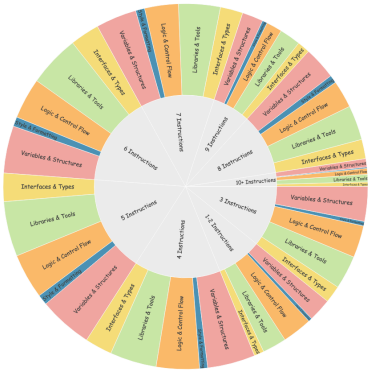

Figure 8: The distributions of schema category and the number of instructions.

It can be found that our dataset exhibits a robust hierarchy of complexity. The inner circle denotes the number of concurrent constraints per problem, ranging from 1 to over 10. Notably, a significant portion of the dataset comprises problems with 4 to 9 instructions, indicating a high level of task difficulty. The outer ring reveals a balanced distribution across functional domains, such as *Logic & Control Flow* and *Interfaces & Types*, ensuring that the model undergoes comprehensive training across diverse coding dimensions.

Examples We present representative examples of the generated IF coding data across varying complexity levels, specifically featuring 2, 4, 6, and 8 instructions. Each instance comprises a formal problem definition, a list of constraints, example input-output pairs, a function signature, and a reference solution. Note that the foundational instruction, i.e., *'Implement the solution in Python'*, is treated as a global prerequisite and is excluded from the total instruction count.

Example with 2 instructions

Problem Description
Write a function that takes a list of tuples where each tuple contains a name and a mark. The function should compute the total marks for each name by summing their marks, then return the tuple consisting of the name with the highest total marks and their total.

Instruction
1. Implement the solution using Python.
2. It is forbidden to use the built-in 'max' function in your code.
3. The code must make use of list comprehension.

Example Input and Output
Example Input 1: [('Juan Whelan',90),('Sabah Colley',88),('Peter Nichols',7),('Juan Whelan',122),('Sabah Colley',84)]
Example Output 1: ('Juan Whelan', 212)
Example Input 2: [('Juan Whelan',10),('Sabah Colley',20),('Peter Nichols',30),('Juan Whelan',40),('Sabah Colley',50)]
Example Output 2: ('Sabah Colley', 70)

Function Signature

[4] https://tree-sitter.github.io/

```
def max_aggregate(stdata):
    pass
```

Solution

```
def max_aggregate(stdata):
    aggregated = [
        (name, sum(mark for n, mark in stdata if n == name)) \
            for name in {n for n, _ in stdata}
    ]
    max_name = None
    max_total = 0
    for name, total in aggregated:
        if max_name is None or total > max_total:
            max_name = name
            max_total = total
    return (max_name, max_total)
```

Example with 4 instructions

Problem Description
Write a Python function to check whether the given number can be represented as the sum of two squares of positive integers.

Instruction
1. Develop the solution using Python.
2. Define a variable called found and set its initial value to False in your code.
3. Utilize the set data structure in your implementation.
4. Include a switch (or match/case) statement within the code.
5. Make sure to import the math library in your code.

Example Input and Output
Example 1:
Input: 25
Output: True (since $3^2 + 4^2 = 25$)
Example 2:
Input: 24
Output: False

Function Signature

```
def sum_Square(n):
    pass
```

Solution

```
import math

def sum_Square(n):
    found = False
    squares = set()
    i = 1
    while i <= math.isqrt(n):
        square = i * i
        squares.add(square)
        remainder = n - square
        match remainder:
            case x if x in squares and x != 0:
                found = True
                break
        i += 1
```

```
    return found
```

Example with 6 instructions

Problem Description
Write a function to determine if a given number is abundant. An abundant number is a number for which the sum of its proper divisors (excluding the number itself) is greater than the number.

Instruction
1. Implement the solution using Python.
2. Ensure every variable name in your code adheres to the snake_case naming convention.
3. Do not import the math library in your code.
4. Make use of list comprehension in your implementation.
5. Avoid using any for loops in your code.
6. Incorporate the set data structure in your solution.
7. Include exactly one while loop in your code.

Example Input and Output
Example Input 1: 12
Example Output 1: True
Example Input 2: 15
Example Output 2: False

Function Signature

```
def check_abundant(n):
    pass
```

Solution

```
def check_abundant(n):
    divisors = set()
    i = 1
    while i * i <= n:
        if n % i == 0:
            divisors.add(i)
            if i != n // i:
                divisors.add(n // i)
        i += 1
    proper_sum = sum([d for d in divisors if d != n])
    return proper_sum > n
```

Example with 8 instructions

Problem Description
Given a list of toy types, return true if a permutation of the list could form a palindrome sequence (reads the same forwards and backwards) and false otherwise.

Instruction
1. Implement the solution using Python.
2. The code must include exactly one list comprehension and one while loop.
3. Ensure your code does not define a variable with the name "index".
4. It is mandatory to incorporate a switch (or match/case) statement in the code.
5. The solution must make use of a generator expression.
6. You must employ the built-in ‘collections.Counter‘ function.
7. A global variable named ‘palindrome_check_enabled‘ must be defined in the code.
8. Full type annotations (type hints) are required for every function and variable.

9. The built-in ‘sum‘ function is strictly prohibited from use in the solution.

Example Input and Output
Example 1:
Input: s = "code"
Output: false

Example 2:
Input: s = "aab"
Output: true

Example 3:
Input: s = "carerac"
Output: true

Function Signature

```
def canPermutePalindrome(s: str) -> bool:
    pass
```

Solution

```
from collections import Counter
from typing import Generator, Iterable, Iterator

palindrome_check_enabled: bool = True

def canPermutePalindrome(s: str) -> bool:
    global palindrome_check_enabled
    if not palindrome_check_enabled:
        return False
    chars: list[str] = [c for c in s]
    char_count: Counter[str] = Counter(chars)
    odd_counts: Generator[int, None, None] = (v % 2 for v in
        char_count.values())
    odd_count_total: int = 0
    odd_iter: Iterator[int] = iter(odd_counts)
    while True:
        try:
            val: int = next(odd_iter)
            match val:
                case 1:
                    odd_count_total += 1
                case _:
                    pass
        except StopIteration:
            break
    return odd_count_total < 2
```

D Additional Experiments

Java Extension. To verify that IFCodeEvolve generalizes beyond Python, we adapt the schema library to Java as described in Appendix C.1. For the seed dataset, we integrate three public Java sources: MBJP (Li et al., 2025), the Java subset of NaturalCodeBench (Zhang et al., 2024b), and ClassEval-T (Xue et al., 2025), comprising 857, 85, and 43 problems, respectively, for a total of 985 instances spanning basic logic, real-world software development, and class-level design. We apply the same filtering and augmentation pipeline as in Python without language-specific tuning. For evaluation, we use the Java subsets of IFEvalCode (Yang et al., 2025) and CodeIF (Yan et al., 2025). As shown in Table 9, IFCodeEvolve yields consistent improvements on both base actors, demonstrating that the framework generalizes across programming languages.

Table 9: IF-Coding benchmarks (Java).

	IFEvalCode		CodeIF	
	Inst.	Prompt	Inst.	Prompt
GLM-5.2	0.7928	0.7059	0.5512	0.3258
GPT-OSS-20B	0.8481	0.5343	0.8407	0.4051
GPT-OSS-120B	0.9102	0.6863	0.8408	0.4844
Seed-OSS-36B	0.8808	0.7010	0.9247	0.5326
Qwen2.5-Coder-32B	0.8638	0.5637	0.8527	0.3229
Seed-2.0-thinking	0.9415	0.8333	0.9282	0.5977
Qwen2.5-Coder-7B	0.7115	0.2500	0.6346	0.0765
+ IFCODEEVOLVE	0.8395	0.4853	0.8986	0.3881
Seed-Coder-8B	0.7917	0.3971	0.7912	0.1898
+ IFCODEEVOLVE	0.8471	0.5049	0.8947	0.3909

Table 10: Ablation study on core modules.

	IFEvalCode		CodeIF	
	Inst.	Prompt	Inst.	Prompt
Qwen2.5-Coder-7B + IFCODEEVOLVE	0.8701	0.6143	0.8435	0.3007
w/o MCTS	0.8542	0.5952	0.8388	0.2989
w/o Actor Evolving	0.8611	0.5930	0.8271	0.2672
w/o Instruction Mutation	0.8616	0.5907	0.8399	0.2917
w/o Instruction Composition	0.8677	0.5988	0.8410	0.2890
w/o Instruction Evolving	0.8589	0.5910	0.8307	0.2790
Seed-Coder-8B + IFCODEEVOLVE	0.8519	0.6095	0.8520	0.3333
w/o MCTS	0.8514	0.5905	0.8382	0.2902
w/o Actor Evolving	0.8433	0.5870	0.8421	0.2903
w/o Instruction Mutation	0.8469	0.5900	0.8400	0.2999
w/o Instruction Composition	0.8456	0.5823	0.8455	0.3023
w/o Instruction Evolving	0.8373	0.5758	0.8301	0.2895

Reward Function We investigate the impact of reward shaping by ablating the reward function design on the same training corpus, as shown in Figure 9(a). Compared to the dense reward (Eq. 6), we examine two distinct variants:

- Sparse Reward: We remove the dense shaping term, providing instruction feedback only upon perfect compliance.

$$r = r_{\text{func}} + \mathbb{I}(r_{\text{IF-dense}} = 1) + 2 \cdot (r_{\text{func}} \cdot r_{\text{IF-dense}}) \tag{7}$$

- Conditional Reward: We employ a gated formulation where instruction adherence is rewarded only if the code is functionally correct.

$$r = r_{\text{func}} \cdot \big(1 + r_{\text{dense}} + 2.0 \cdot \mathbb{I}(r_{\text{dense}} = 1)\big) \tag{8}$$

As shown in Figure 9, we present the learning curves on entropy and reward across the training on `Seed-Coder-8B` (a) and `Qwen2.5-Coder-7B` (b). The Conditional Reward exhibits the poorest performance due to its strict gating mechanism, which masks instruction adherence signals whenever functional correctness fails. This creates reward sparsity during early training. In contrast, the Dense/Sparse Rewards provides continuous feedback independent of functional success, facilitating a smoother optimization process.

Ablation Study We conduct an ablation study to assess the effectiveness of the core components of IFCODEEVOLVE. We consider five variants: (1) *w/o MCTS*, where MCTS-guided search is replaced by uniform random sampling; (2) *w/o Actor Evolving*, where the actor is frozen and the full volume of data is generated in a single round; (3) *w/o Instruction Mutation* and (4) *w/o Instruction Composition*, where only one of the two schema evolution operators is disabled; and (5) *w/o Instruction Evolving*, where both operators are disabled and the schema library remains fixed throughout. As shown in Table 10, removing any component lowers performance on all metrics for both base actors, confirming that each component contributes to the final performance.

Schema evolution plays a particularly important role. Disabling it entirely causes a larger drop than disabling either operator alone on seven of the eight metrics, indicating that composition and mutation are complementary. The effect is most pronounced for the stronger actor, `Seed-Coder-8B`, for which removing schema evolution leads to the largest drop among all ablated components on every metric, e.g., by 3.37 and 4.38 points on the Prompt metrics of IFEvalCode and CodeIF, respectively. Beyond benchmark scores, schema evolution also increases the diversity of the synthesized instructions (Figure 5 and Table 11) and improves generalization to schemas unseen during synthesis (Table 12).

Diversity Analysis To directly measure the contribution of schema evolution, we conduct a matched-budget ablation with `Qwen2.5-Coder-7B` as the actor, comparing the full method against three variants: a fixed schema library, composition only, and mutation only. We then measure the diversity of the instructions in each synthesized corpus. An instruction type is identified by its schema together with its instantiated parameter tuple, so linguistic paraphrases are not counted as new types. Let p_i denote the proportion of the i-th instruction type among the n instruction occurrences, and f_1 the number of types observed exactly once. We report:

Table 11: Diversity of the instructions synthesized under a matched budget with different schema evolution strategies, using `Qwen2.5-Coder-7B` as the actor.

	#Types ↑	Hill-1 ↑	Hill-2 ↑	Coverage ↓
Fixed Schema	313	111.7	51.5	0.8685
Mutation Only	354	144.6	66.4	0.8652
Composition Only	391	167.5	79.1	0.8335
IFCODEEVOLVE	547	243.0	92.2	0.7627

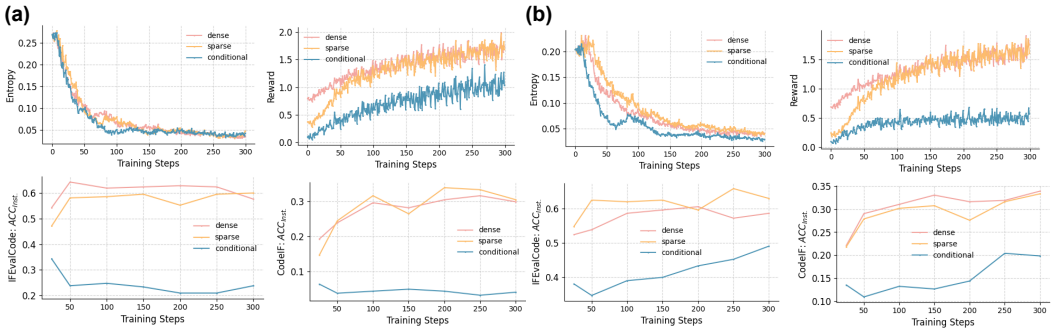

Figure 9: Learning curves of different RL reward functions using `Qwen2.5-Coder-7B` **(a)** and `Seed-Coder-8B` **(b)** as base models

- *#Types*: the number of distinct instruction types.
- *Hill-1* $= \exp(-\sum_i p_i \ln p_i)$ and *Hill-2* $= 1/\sum_i p_i^2$ (Hill, 1973): the effective number of equally frequent instruction types, where Hill-2 places more weight on frequent types. Higher values indicate greater diversity.
- *Coverage* $= 1 - f_1/n$ (Good, 1953): the estimated probability that the next sampled instruction belongs to an already observed type. Lower values indicate that the instruction space is further from being exhausted.

As shown in Table 11, full schema evolution yields $2.18\times$ the Hill-1 effective number of instruction types of the fixed library, together with the lowest coverage. Both operators contribute: composition only and mutation only reach $1.50\times$ and $1.30\times$ the Hill-1 diversity of the fixed library, respectively, while combining them performs best on all metrics. Together with the round-over-round shift of top-ranked schemas (Figure 5), this indicates that schema evolution keeps expanding the instruction space rather than recycling a fixed set of patterns.

Evolution Pattern Analysis We analyze the evolution patterns of the schema library across rounds, using `Seed-Coder-8B` as the actor. From Round 1 to Round 2, 3 schemas are removed, and 6 are added (3 composed, 3 mutated); from Round 2 to Round 3, 2 are removed, and 5 are added (3 composed, 2 mutated). Three effective evolution patterns emerge:

- First, the library can strengthen existing constraints rather than simply preserving them. For example:
  - "Your code {`should_import`} import the library {`library_name`}" is evolved into "Your code {`should_import`} import the library {`library_name`}, and the code must contain at least two import statements."
  - "All variable names in your code must follow the {`convention`} naming convention." is strengthened into "All variable names in your code must follow the {`convention`} naming convention and contain at least one digit."
- Second, evolution adds structural context to previously shallow constraints. For instance:
  - "Your code must define a variable named {`variable_name`} initialized with the value {`initial_value`}." is evolved into "Your code must define a variable named {`variable_name`} initialized with the value {`initial_value`} inside a nested block (e.g., inside an if statement, for loop, while loop, try-except block, or with statement)."

Table 12: Instruction compliance on instructions instantiated from held-out schemas.

	Original	w/o Schema Evolution	IFCODEEVOLVE
`Qwen2.5-Coder-7B`	0.3972	0.4831	0.5061
`Seed-Coder-8B`	0.4581	0.5411	0.5526

Table 13: Mean nearest-neighbor TF-IDF similarity between benchmark problems and the synthesized training corpus.

	IFEvalCode	CodeIF
`Qwen2.5-Coder-7B`	0.1257	0.1022
`Seed-Coder-8B`	0.1226	0.0800

  - "The length of variable names {`comparison`} exceed {`length`} characters." is evolved into "The length of variable names from both assignment statements and loop targets {`comparison`} exceed {`length`} characters."

- Third, the composed schemas reveal a clear cross-dimension composition pattern: composition not only (1) merges similar constraints, such as "naming convention + prefix", and "if-else count + loop count", but also (2) combines constraints from different aspects of code generation, such as "forbidden built-ins + forbidden loops", "forbidden loops + no import", and "comment count + type hints". This pattern suggests that composition is mainly used to create harder interaction effects across multiple coding dimensions.

Held-out Schema Generalization To examine whether the synthesized data teaches transferable instruction-following capability rather than compliance with a fixed checklist of constraint types, we conduct a held-out-schema experiment. We randomly hold out 6 schemas from the library and exclude them from synthesis. The evaluation set consists of the IFCODEBENCH problems that contain at least one instruction instantiated from the held-out schemas; we exclude every problem whose seed task overlaps with the training corpus to prevent task leakage, and score compliance only on the instructions instantiated from the held-out schemas. To isolate the contribution of schema evolution, we further compare against IFCODEEVOLVE without schema evolution, keeping the rest of the pipeline unchanged.

As shown in Table 12, IFCODEEVOLVE substantially improves compliance on constraint types never observed during synthesis for both actors, indicating that the learned capability generalizes beyond the schema library. Schema evolution accounts for part of this gain, improving over the variant without it by 2.30 and 1.15 percentage points for `Qwen2.5-Coder-7B` and `Seed-Coder-8B`, respectively, which shows that its benefit is not limited to corpus-level diversity.

Contamination Analysis In addition to the exact n-gram overlap analysis in Table 4, we perform a TF-IDF nearest-neighbor check, which is also sensitive to partially rewritten problems and constraints. Each problem is represented as an L2-normalized TF-IDF vector over ASCII word tokens and CJK character bigrams, and for each benchmark problem we compute its maximum cosine similarity to the synthesized training corpus. As shown in Table 13, the mean maximum similarity stays below 0.13 for both actors and both benchmarks, providing further evidence against substantial lexical reuse between our synthesized data and the evaluation benchmarks.

E IFCODEBENCH

E.1 STATISTICS AND ANNOTATION

Source Composition As described in Section 4.3, IFCODEBENCH is curated from the 4,241 problems synthesized with `Seed-1.6` as the generator and `Seed-Coder-8B` as the actor (Appendix C.2). Its 530 problems originate from MBPP (75.5%), LeetCode (21.1%), and ClassEval (3.4%). `Seed-1.6` is used for generation, LLM-based filtering, and LLM-based evaluation.

Schema Statistics The instructions in IFCODEBENCH are instantiated from initial (62.7%), composed (20.9%), and mutated (16.4%) schemas. Table 14 reports the distribution over schema categories.

Annotation Protocol All instances were verified by 13 annotators, all professional developers with at least three years of software engineering experience, none of whom are authors of this paper.

Table 14: Distribution of schema categories in IFCODEBENCH.

Category	Structural Control	Variable	Function	Interface	Global	File
#	808	626	521	302	244	129
%	30.7	23.8	19.8	11.5	9.3	4.9

Each candidate was independently labeled by at least three annotators using the six-criterion rubric in Appendix E.2. Annotation ran over five days, averaging 5–10 minutes per instance. To handle disagreement, a candidate was retained only if every assigned annotator passed it on all four binary sanity criteria.

E.2 RUBRIC

We detail the evaluation rubric employed to curate IFCODEBENCH, comprising six distinct criteria. These are divided into two categories: Sanity Metrics and Quality Metrics. The first four, i.e., *Consistency, Redundancy, Validity,* and *Alignment*, serve as binary filters; any candidate failing a single criterion (scoring 0) is discarded. The remaining two, including *Transformation* and *Value*, assess the substantive difference between the vanilla and constrained solutions. Our curation pipeline begins with an automated phase where `Seed-1.6` filters candidates, retaining only those meeting the sanity requirements alongside thresholds of Transformation ≥ 2 and Value ≥ 2. These candidates then undergo rigorous manual inspection using the same rubric. Ultimately, IFCODEBENCH consists of 530 gold-standard instances that satisfy all four sanity checks and achieve a maximum score of Value $= 3$. The scoring strategies and representative examples for each criterion are provided below.

- Consistency: Each instruction must not conflict with the problem description or other instructions.
  - Scoring: 0 / 1
  - Negative Example 1: "Ensure every variable name in your code adheres to the camelCase naming convention." AND "The code must avoid using any intermediate or temporary variables.".
  - Negative Example 2: "Import the 'math' library in your code." AND "Do not use the 'math.pow' built-in function".
- Redundancy: The instructions must be unique and necessary.
  - Scoring: 0 / 1
  - Negative Example 1: "All variable names must start with the prefix 'num' and adhere to the camelCase naming convention." AND "All variable names must follow camelCase and include at least one digit.".
  - Negative Example 2: "The code must include type annotations for every function and variable, and you are required to use the built-in 'collections.Counter' function." AND "Ensure the code has exactly two lines of comments and provides type hints for all functions and variables.".
- Validity: Instructions must be verifiable.
  - Scoring: 0 / 1
  - Negative Example 1: "The code must be elegant.".
  - Negative Example 2: "The code must be highly readable and use meaningful variable names.".
- Alignment: The provided solution must satisfy all instructions.
  - Scoring: 0 / 1
  - Negative Example 1: "Implement a function to calculate the factorial of n using recursion" AND solution:

```
def factorial(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

  - Negative Example 2: "The code cannot use any built-in sorting functions" AND solution:

```
def sort_array(arr):
    return sorted(arr)
```

- Transformation: Evaluate the extent of structural modification between the vanilla solution (generated without specific constraints) and the constrained solution.
  - Scoring: 1 / 2 / 3 (1: Surface, 2: Medium, 3: High)
  - Surface-level: Cosmetic changes only. The vanilla solution:

```
import heapq
def merge_sorted_list(num1,num2,num3):
  num1 = sorted(num1)
  num2 = sorted(num2)
  num3 = sorted(num3)
  result = heapq.merge(num1,num2,num3)
  return list(result)
```

    The constrained solution:

```
import heapq
def merge_sorted_list(num1, num2, num3):
    sorted_num1 = sorted(num1)
    sorted_num2 = sorted(num2)
    sorted_num3 = sorted(num3)
    sorted_result = heapq.merge(sorted_num1, sorted_num2,
        sorted_num3)
    return list(sorted_result)
```

  - Medium-level: Apply different data structure, control flow, but still the same logical flow. The vanilla solution:

```
def text_match_three(text):
    for i in range(len(text) - 3):
        if text[i] == 'a' and text[i+1] == 'b' and text[i+2] == 'b'
            and text[i+3] == 'b':
            return 'Found a match!'
    return 'Not matched!'
```

    The constrained solution:

```
import re
def text_match_three(text):
    patterns = 'ab{3}?'
    if re.search(patterns, text):
        return 'Found a match!'
    else:
        return('Not matched!')
```

  - High-level: Paradigm Shift / Algorithmic Change. The vanilla solution:

```
def permutation_coefficient(n, k):
    P = [[0 for i in range(k + 1)] for j in range(n + 1)]
    for i in range(n + 1):
        for j in range(min(i, k) + 1):
            if (j == 0):
                P[i][j] = 1
            else:
                P[i][j] = P[i - 1][j] + (j * P[i - 1][j - 1])
            if (j < k):
                P[i][j + 1] = 0
    return P[n][k]
```

    The constrained solution:

```
from functools import reduce
def permutation_coefficient(N, K):
    if K == 0:
        return 1
```

```
    if K > N:
        return 0
    return reduce(lambda X, Y: X * Y, range(N, N - K, -1))
```

- Value: The goal is to assess whether the instruction leads to meaningful logical changes, algorithmic optimizations, or paradigm shifts in the code, rather than simply making changes for the sake of change (or making things worse).
  - Scoring: 1 / 2 / 3 (1: Negative, 2: Neutral, 3: Positive)
  - Negative value: Instruction forces the model to generate code that is more difficult to read, less secure, and technically unsound. The vanilla solution:

```
def extract_string(str, l):
    result = [e for e in str if len(e) == l]
    return result
```

    The constrained solution:

```
def data_Recursive_Len(data_String):
    if not data_String:
        return 0
    return 1 + data_Recursive_Len(data_String[1:])

data_Filter_Check = lambda data_String, data_Len:
     data_Recursive_Len(data_String) == data_Len

def extract_string(data_InputList, data_TargetLength):
    data_FilteredResult = list(
        e for e in data_InputList if data_Filter_Check(e,
            data_TargetLength)
    )
    return data_FilteredResult
```

  - Neutral value: The instructions led to changes in the code's structure, but the core logic, complexity, and engineering quality remained essentially unchanged. The vanilla solution:

```
def extract_index_list(l1, l2, l3):
    result = []
    for m, n, o in zip(l1, l2, l3):
        if (m == n == o):
            result.append(m)
    return result
```

    The constrained solution:

```
def extract_index_list(l1, l2, l3):
    result = []
    index = 0
    min_len = min(len(l1), len(l2), len(l3))
    while True:
        if index >= min_len:
            break
        m, n, o = l1[index], l2[index], l3[index]
        if m == n == o:
            result.append(m)
        index += 1
    return result
```

  - Positive value: The instruction led to a solution that is educationally valuable, algorithmically alternative, or engineering-robust. Although the new solution may not necessarily be better than the original one, it demonstrates a different way of thinking. The vanilla solution:

```
def permutation_coefficient(n, k):
    P = [[0 for i in range(k + 1)] for j in range(n + 1)]
    for i in range(n + 1):
        for j in range(min(i, k) + 1):
            if (j == 0):
```

```
                P[i][j] = 1
            else:
                P[i][j] = P[i - 1][j] + (j * P[i - 1][j - 1])
            if (j < k):
                P[i][j + 1] = 0
    return P[n][k]
```

The constrained solution:

```
from functools import reduce
def permutation_coefficient(N, K):
    if K == 0:
        return 1
    if K > N:
        return 0
    return reduce(lambda X, Y: X * Y, range(N, N - K, -1))
```

F PROMPT

Actor prompt

As a programming assistant, your task is to generate code snippets based on the user question and instructions given below:

Requirements
- Make sure you follow the user instructions. If the instruction says to use a specific language or a specific method, use exactly that.
- Your output should be a valid code snippet in the programming language indicated in the question or the instructions.
- Remember to import any necessary libraries or modules if needed.
- Remember to import Typing if the signature contains type hints.

Output Format
The output should only be a valid code snippet without any explanations, comments, or text outside the code.

Problem
{prompt}

LLM evaluator for instruction following

Role
You are an expert Code Compliance Auditor. Your task is to verify whether a model-generated code solution strictly follows a specific set of user instructions.

Workflow
You must follow these two steps strictly:

Step 1: Step-by-Step Analysis
- Iterate through each instruction in the provided list.
- For each instruction, examine the "Model-generated Code" to see if the requirement is met.
- Briefly explain your reasoning for each instruction.
- Conclude each analysis with a "Yes" or "No".

Step 2: Final Output Generation - Collect your "Yes" or "No" verdicts into a Python-style list.
- Ensure the list length is exactly the same as the number of instructions.
- Wrap this list inside ‘<answer>’ and ‘</answer>’ tags.

- The content inside ‘<answer>’ must be **only** the list (e.g., ‘[‘Yes’, ‘No’]’), without any reasoning text.

Evaluation Criteria
- **Strict Adherence:** If a specific language, library, algorithm complexity, or naming convention is requested, it must be present.
- **Syntax:** The code must be syntactically valid for the requested language.
- **Order:** The i-th element in your result list must correspond to the i-th instruction.

Input Data
Coding Question
{question}

Instructions
{instruction}

Model-generated Code
{code}

Execution
Please begin your Step 1 Analysis now, followed immediately by the Step 2 Final Output.

Proof-by-construction by generator

You are an expert Python coding challenge designer and data synthesizer. Your task is to "mutate" a given programming problem (Seed Question) by applying a specific **instruction**, thereby generating a new, more challenging version of the problem.

Input Format
You will receive an XML snippet containing:
1. ‘<question>’: The original problem description, existing instructions, the reference solution (Code), programming language, and language.
2. ‘<Mutations>’: A list of ‘<Mutation>’ tags. Each mutation is a template for a new instruction and a list of ‘<params>’ required to instantiate that template.

Workflow (Step-by-Step)
Before generating the final XML output, you must perform the following reasoning steps inside a ‘<thought>’ tag:
1. Analyze the Original Code:
- Understand the algorithm, complexity, and existing instructions of the seed code.

2. Strategic Parameter Selection and Applicability Check:
- Iterate through **each** provided ‘<Mutation>’ in the list.
  - For each candidate, evaluate two criteria:
    - **Compatibility**: Does this mutation make sense for this problem and the current instructions?
    - **Challenge Level**: How much does this force a refactor?
  - **Selection Strategy**:
    - Discard Incompatible mutations.
    - If multiple have the same challenge level, prioritize the one that aligns best with Pythonic best practices or specific algorithmic concepts.

3. Parameter Instantiation:
- For the selected mutation, look at its ‘<params>’.
  - If ‘<Mutation>/<params>’ is empty, skip parameter selection and proceed directly to conflict detection. The instruction text is fixed.

- If ‘<Mutation>/<params>’ is not empty:
  - **Maximize Challenge**: Choose parameter options that contradict the *current* implementation (e.g., if code uses a ‘for‘ loop, and options are ‘[’while’, ’recursion’]‘, choose ‘recursion‘ if valid).
  - **Feasibility**: Ensure the chosen parameters allow for a valid solution.

4. Conflict Detection:
- Ensure the selected mutation and its parameters do not contradict the original ‘<instruction>’.
- Such as ’Your code must utilize exactly 1 list comprehension.’ and ’Your code must not use any for loops.’, which are incompatible with each other.

5. Refactor Code (Only if Successful):
- If successful, rewrite the reference code to **strictly adhere** to the new instantiated constraint.
- The modified code must produce the exact same output for the same inputs as the original code.

6. Synthesize Output:
- Generate the ‘<success>’ tag first.
  - If ‘<success>false</success>’, **STOP** after closing the tag. Do not generate params or question.
  - If ‘<success>true</success>’, generate ‘<instantiated_params>’ and the modified ‘<question>’.
    - If the input ‘<Mutation>/<params>’ was empty, the tag <instantiated_params> must also be empty (i.e., ‘<instantiated_params></instantiated_params>’).

Output Format
Return the result strictly in XML format. Do not include markdown code block markers (like “‘xml).

```
<output>
  <thought><![CDATA[1. Analysis: [Analysis of original code]
2. Evaluation:
   - Mutation ID 1: [Compatibility: Yes/No] | [Challenge: Low/Med/
       High] | [Reasoning]
   - Mutation ID 2: ...
   - Decision: Selected Mutation ID [X] because...
3. Parameter Selection: [Reasoning for chosen params]
4. Refactoring Strategy: [How the code will change]]]></thought>
  <success>[true/false]</success>
  <instantiated_params>
    <param>
      <name>[Parameter Name, e.g., variable_name]</name>
      <value>[Selected Value, e.g., total_score]</value>
    </param>
  </instantiated_params>
  <question>
    <question_desc>[Original description. Modify only if the mutation
        fundamentally changes the output format, otherwise keep as
       is.]</question_desc>
    <instruction>
  [Existing instructions]
  [New instantiated instruction]
    </instruction>
    <code><![CDATA[
  [The full, modified Python code]
    ]]></code>
    <function_signature><![CDATA[ function signature that should be
       aligned with the code ]]></function_signature>
    <lg>[en/zh]</lg>
```

```
    <programming_language>python</programming_language>
  </question>
</output>
```

Constraints
- Do **not** alter the core algorithmic intent (e.g., if the problem is "Sum of list", do not change it to "Average of list").
- The ‘<instruction>’ field must contain **all** previous instructions plus the new one, separated by newlines and numbering.
- The code must be wrapped in ‘<![CDATA[...]]>’.

Instruction mutation

You are an expert in program analysis and synthetic coding data generation.

Your task is to EVOLVE an existing instruction into a STRICTLY STRONGER, AST-CHECKABLE instruction that INVALIDATES ALL GIVEN EXAMPLES, while still SUBSUMING the original instruction.

The goal is to generate a higher-difficulty, verifier-backed instruction that exposes genuine unconstrained syntactic degrees of freedom.

Input
You will receive an XML describing an existing instruction, including:
- type
- instruction template
- checking function

Multiple XML examples of solved coding problems, containing:
- question_desc
- instruction(s)
- example_input_and_output
- code
- instantiated_params

All provided examples are VALID solutions to the ORIGINAL instruction.

Goals
You MUST design an EVOLVED instruction such that:

1. Subsumption
- Any program that satisfies the EVOLVED instruction MUST also satisfy the ORIGINAL instruction.
- You must explicitly justify this subsumption.
- The evolved instruction MUST introduce at most ONE new AST-level constraint beyond those already enforced by the original instruction.
- The evolved instruction shouldn’t be overly complicated

2. Examples-as-Negative
- ALL provided example programs MUST FAIL the EVOLVED instruction.
- These failures must arise from a GENERAL, STRUCTURAL, AST-level constraint.
- You MUST NOT reference specific identifiers, literals, or fingerprints unique to the examples.

3. AST-Checkability
- The evolved instruction MUST be checkable using static AST analysis.
- No runtime execution, no I/O, no performance, no semantic reasoning.

4. Generalization
- The evolved instruction MUST generalize beyond the given examples.
- You are FORBIDDEN from writing constraints that merely exclude the examples without structural meaning.

You MUST NOT:
- Mention or encode specific variable names, constants, or literals that appear only in the examples.
- Refer to line counts, whitespace, formatting, or comments.
- Use vague or semantic constraints such as: "readable", "clean", "efficient", "reasonable", "well-structured".
- Encode example-specific signatures or hashes.
- Introduce constraints that can be satisfied or violated trivially.

Output format (MUST be valid XML, no extra text):
The output should only be a valid XML document without any extra text:

```xml
<output>
  <thought><![CDATA[
    A detailed explanation covering:
    - Interpretation of the original instruction
    - How the examples satisfy it
    - What syntactic freedom is being restricted
    - Why the evolved instruction is strictly stronger
    - Why it subsumes the original
    - Why all provided examples fail it
    ]]></thought>

  <instruction><![CDATA[
    The evolved instruction template (may contain placeholders)
  ]]></instruction>

  <function>
    <name>FUNCTION_NAME</name>
    <params>
      <name>param_name</name>
      <type>string | integer | boolean | enum</type>
      <options>...</options> <!-- only if type=enum -->
    </params>
    <params>
      <name>param_name</name>
      <type>string | integer | boolean | enum</type>
      <options>...</options> <!-- only if type=enum -->
    </params>
    ...
    <impl><![CDATA[
def FUNCTION_NAME(tree, code_str, param1, param2, **kwargs):
    # AST-based deterministic checking logic
    return True or False
]]>
    </impl>
  </function>

  <positive_cases>
    <case>
      <instantiated_params>
        <!-- a valid parameter instantiation -->
      </instantiated_params>
      <code>
      <![CDATA[
```

```
# Python code that SATISFIES the evolved instruction under the given
    parameter instantiation
]]>
      </code>
    </case>
    <case>
      <instantiated_params>
        <!-- a valid parameter instantiation -->
      </instantiated_params>
      <code>
      <![CDATA[
# Python code that SATISFIES the evolved instruction under the given
    parameter instantiation
]]>
      </code>
    </case>
  </positive_cases>
</output>
```

Problem rephrase

You are an expert Data Augmentation Specialist for code generation tasks. Your goal is to rewrite coding instructions to increase linguistic diversity while STRICTLY preserving the original semantic meaning and logic.

Task
You will receive a list of coding instructions (formatted as a numbered list). You must generate a rephrased version of this list. The new instructions should sound natural, diverse, and human-written.

Transformation Rules
1. **Rephrase**: This is your main tool. Use synonyms, change sentence structures, and vary the tone to express the exact same requirement.
2. **Combine**: Merge two or multiple instructions into a single instruction.
3. **Preserve Logic**:
- Do NOT change any values (e.g., if it says "length < 5", do not output "length <= 5").
- Do NOT change the requirement type (e.g., "must use" cannot become "can use").
- Do NOT omit any constraints.
4. **Format**: The output must be a numbered list separated by newlines.