CodeEvo: Interaction-Driven Synthesis of Code-centric Data through Hybrid and Iterative Feedback

Anonymous ACL submission

Abstract

Acquiring high-quality instruction-code pairs is essential for training Large Language Models for code generation. Manually curated data is expensive and limited in scale, motivating the development of code-centric synthesis methods. Yet, current approaches often rely on predefined heuristics, resulting in synthetic data that is ungrounded, repetitive, or simplistic. We propose *CodeEvo*, a framework inspired by collaborative programming that employs two interacting LLM agents. A Coder generates and refines solutions, while a Reviewer directs the synthesis process. To overcome the limitations of simple heuristics, the Reviewer first constructs a Schema, a structured blueprint that explicitly plans the logic, constraints, and complexity of a new instruction prior to its generation. This planning process is complemented by a hybrid feedback mechanism that combines compiler determinism with the agent's semantic evaluation, ensuring rigorous quality control. Extensive experiments demonstrate that models fine-tuned on *CodeEvo* data significantly outperform established baselines across code generation benchmarks. In-depth analyses further provide insights into effective code-centric data synthesis. Code and data are available at `https://anonymous.4open.science/r/Code_Evo/`.

1 Introduction

The rapid development of Large Language Models (LLMs) has significantly advanced code intelligence (Sun et al., 2024a), powering applications ranging from line-level code completion to competition-level problem solving. To further enhance their performance on code generation, it is essential to train these models with complex, diverse, and grounded instruction-code pairs (Zan et al., 2023). While manually curated data serve as ideal resources, their collection is labor-intensive, difficult to scale, and gradually exhausted (Huang et al., 2024b). These limitations have stimulated growing interest in constructing code-centric synthetic data with minimal human intervention.

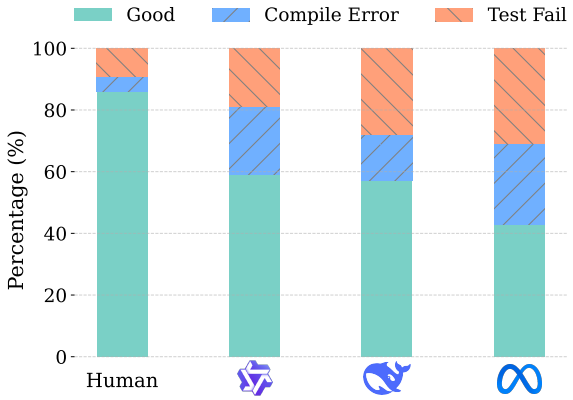


Figure 1: Comparison of synthesized code data quality across human examples and code produced by Qwen2.5-Coder-32B, DeepSeek-V3, and Llama-3.1-8B.

After early attempts that leverage symbolic augmentation over existing code references (Wang et al., 2017; Feng et al., 2018), recent research has shifted toward using LLMs to automatically generate instruction-code pairs. These methods, such as Self-Instruct (Wang et al., 2023) and Evol-Instruct (Luo et al., 2024b), aim to bootstrap massive data using powerful models and predefined heuristics. While these approaches enable data construction, they often fall short in ensuring semantic correctness and executability (Liu et al., 2023).

As shown in Figure 1, we sample instruction-code pairs synthesized by various (Code)LLMs using Evol-Instruct heuristics (Xu et al., 2024a). Many of these samples fail to execute or do not pass the provided unit tests, indicating substantial quality gaps. These shortcomings can be attributed to two main factors: (1) instructions are often poorly grounded, leading to vague or inconsistent objec-

tives; and (2) generated codes lack proper validation, due to the absence of robust mechanisms to enforce correctness during synthesis. This motivates a key question: *Can we design a fully automated and reference-free synthesis pipeline that produces well-grounded and executable instruction-code pairs?*

Recently emerging LLM agents have demonstrated strong interactive capabilities (Sun et al., 2023), enabling them to perform tasks through multi-turn interactions and feedback-driven decision making (*e.g.*, collaborative programming; Qian et al., 2024; Wang et al., 2025a). These make them promising candidates for moving beyond vanilla data generation toward verifiable and adaptive synthesis pipelines. Inspired by this potential, we propose *CodeEvo*, an interaction-driven synthesis framework that orchestrates LLM agents to generate high-quality code-centric data. Specifically, a *Coder* agent produces candidate code and tests based on given instructions, while a *Reviewer* agent provides tailored feedback and dynamically constructs new instructions iteratively.

To address the two core challenges in instruction-code synthesis, *CodeEvo* incorporates two key mechanisms: (1) To lift instruction quality, we introduce a schema-driven synthesis process. Guided by task-specific keywords, a Reviewer agent first constructs a Schema, a structured and adaptable blueprint that plans a new problem's logic and complexity before generating instructions. This transforms instruction evolution from a heuristic-based task into a principled design process. (2) To boost functional correctness, we introduce a hybrid feedback loop that iteratively refines solutions by fusing the deterministic verification of a compiler with fine-grained semantic judgment of an LLM agent . The entire pipeline operates with only a small set of seed instructions as input, and requires *no human annotation* or *gold references*, all while being driven by accessible, medium-sized models.

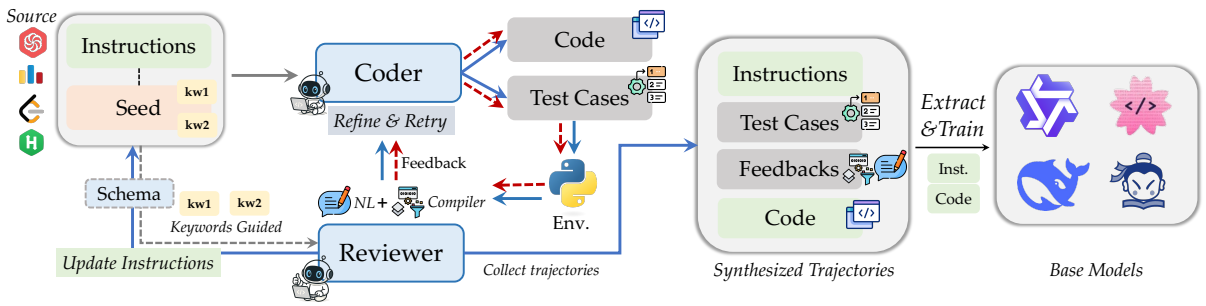
Experiments across multiple backbones and benchmarks demonstrate that *CodeEvo* significantly outperforms established data synthesis methods. Remarkably, *CodeEvo* achieves better performance than competing approaches using several times more data, indicating the superiority of our targeted, feedback-driven synthesis over sheer data volume. Our primary contributions are as follows:

- We propose *CodeEvo*, an interaction-driven synthesis framework that systematically lifts the quality of code-centric synthetic data from both the instruction and code perspectives.
- We introduce an innovative hybrid feedback mechanism that effectively combines the determinism of compiler verification with the generative flexibility of LLM agents.
- Through extensive experiments and analysis, we share insights into the key attributes, including quality, diversity, scalability, and difficulty of synthetic code data.

2 Related Works

LLM-based Agents Interaction. The interactive capabilities of language agents (Sumers et al., 2024), whether with other agents or the environment, have garnered significant attention (Park et al., 2023). These interactions allow for complex problem-solving approaches such as collaboration (Sun et al., 2023; Hong et al., 2024) or role-playing (Li et al., 2023; Islam et al., 2024), which have proven effective in diverse applications like software engineering (Qian et al., 2024; Huang et al., 2024a). Recently, researchers have begun to explore using such interaction for various data generation, including instructions (Mitra et al., 2024), reasoning chains (Yang et al., 2024b), and environment-aware trajectories (Sun et al., 2024b; Hu et al., 2025). Leveraging interaction for flexible and scalable data construction is emerging as a promising direction (Luo et al., 2024a; Khan et al., 2025). This work takes an initial step toward applying interaction to instruction-code synthesis.

Code-centric Data Synthesis. The synthesis of instruction-code data traces back to early symbolic augmentation methods (Wang et al., 2017; Feng et al., 2018), which augment existing code using program transformations. Recent efforts move beyond static heuristics and leverage LLMs to generate instruction-code pairs at scale with prompting (Chaudhary, 2023; Wang et al., 2023; Xu et al., 2025b), as well as through human-in-the-loop interactions (Wei et al., 2024a). Typically, WizardCoder (Luo et al., 2024b) extends Evol-Instruct (Xu et al., 2024a) into code data synthesis, Magicoder (Wei et al., 2024b) and WaveCoder (Yu et al., 2024) derive pairs from open-source code snippets. Further, researchers lay emphasis on executable synthesis (Zheng et al., 2024a; Majumdar et al., 2025; Zeng et al., 2025; Xu et al., 2025a), using code syntax relationships (Wang et al., 2025b) and unit tests (Shao et al., 2025; Ma et al., 2025) to curate code-centric data. However, existing synthe-



(a) Interaction-Driven Synthesis (b) Downstream Application

Figure 2: Overview of the *CodeEvo* framework. The synthesis process begins with seed instructions from different sources. Through continuous interaction, the Coder and Reviewer agents collaboratively construct trajectories of instruction, solution, validation, and refinement. The loop marked with → illustrates the flow of data synthesis, where new instructions are derived and paired with validated code. The loop marked with → captures the validation cycle, incorporating natural language and compiler feedback to guide refinement. Instruction-code pairs are extracted from the validated trajectories and used for downstream model training.

sis methods often rely on pre-defined and limited prompting heuristics, require existing code references, and largely overlook functional correctness.

Compiler Feedback in Code Generation. A key differentiator between symbolic language and natural language is executability (Xu et al., 2024c), with compilers serving as a fundamental verifier (Wang et al., 2022). In the context of LLM-based code generation, leveraging compiler feedback to improve output quality has become an active area of research. Early approaches primarily focused on post-hoc error correction using compiler signals (Lahiri et al., 2022), later evolving to incorporate immediate compiler feedback during generation to improve first-pass correctness (Wang et al., 2022). Static analysis has also been explored to enrich the semantic understanding of code (Ma et al., 2024). More recent efforts have begun decomposing complex generation tasks into subtasks, using compiler feedback for fine-grained optimization (Xu et al., 2024b), debugging (Islam et al., 2025), or repo-level learning (Bi et al., 2024). In addition, compiler feedback has also been leveraged to model preferences (Zhang et al., 2024). While compilers are widely used in the generation stage, their role in the data synthesis pipeline is underexplored. In our synthesis loop, an LLM agent interprets compiler feedback to ensure data quality and guide refinement.

3 *CodeEvo*

We detail the framework of *CodeEvo* in this section, emphasizing its core components and systematic workflow, as illustrated in Figure 2.

3.1 Preliminary

Problem Definition. Given a seed dataset S, containing initial instructions s, each with an associated set of keywords T. The goal is to synthesize an expanded dataset Q. *CodeEvo* consists of two LLM-powered agents:

- **Coder**: Generate candidate solutions and test cases for a given instruction, and refine its solution based on external feedback.
- **Reviewer**: Generate new instructions and evaluate candidate solutions, providing feedback for refinement or data selection.

Seed Instructions. In contrast to prior work that relies on golden code solutions or ready-made test cases, *CodeEvo* requires *only a lightweight set of natural language instructions*, which can originate from any domain where the problem descriptions admit symbolic solutions, such as algorithmic problems from programming platforms, NL2Code training sets, or mathematically structured problems.

3.2 Schema-Guided Instruction Generation

A central challenge in instruction synthesis is maintaining semantic control during evolution. Prior methods often rely on abstract commands (*e.g.*, "make it harder"), which can lead to vague or ungrounded content. To move beyond such abstract heuristics, we introduce Schema-guided instruction generation. This approach transforms synthesis into a principled design process. Guided by task-specific keywords, our Reviewer agent first generates a Schema—a structured blueprint that outlines the logic for combining concepts, the desired complexity, and the overall goal for a new

instruction. By first formulating this design plan before generating the final text, we ensure that new instructions are not only more challenging but also logically coherent and well-grounded.

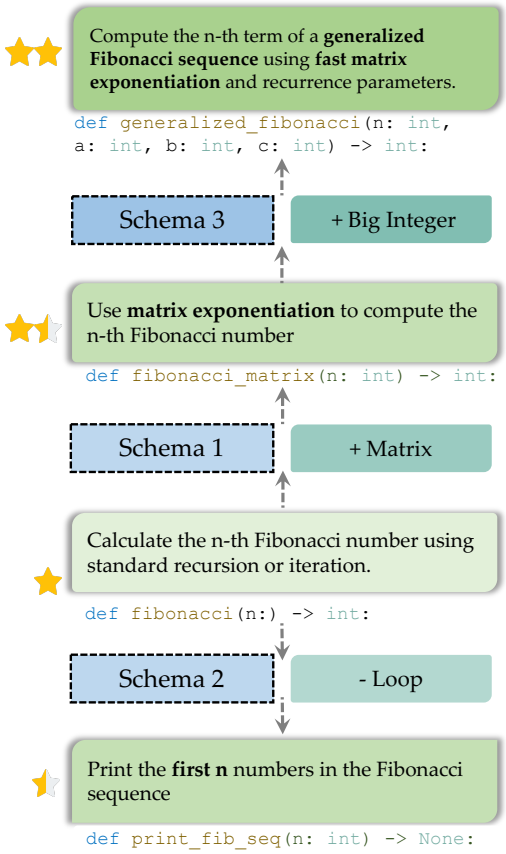


Figure 3: Illustration of transforming a seed into relevant instructions by leveraging schema and keywords.

Specifically, for each seed instruction s and a selected keyword subset $t \subseteq T$, the generation process unfolds in two stages. First, the Reviewer agent formulates a Schema that serves as a detailed plan for the new instruction:

$$\text{Schema} = \text{GenerateSchema}(s, t) \tag{1}$$

Subsequently, it generates the new instruction s' by executing the plan laid out in the Schema:

$$s' = \text{GenInstruction}(s, \text{Schema}) \tag{2}$$

Crucially, this Schema-driven mechanism is inherently bidirectional. The Schema can strategically plan to either integrate keywords for added complexity or selectively omit them to construct a simplified variant, particularly when a task proves intractable for the Coder agent. This structured approach marks a significant shift from rigid prompting heuristics: enabling the generation of novel, diverse instructions while reducing the likelihood of producing unanswerable ones (statistics in Appendix G). An adaptive keyword sampling strategy is detailed in Appendix E.

3.3 Hybrid Feedback for Validation and Refinement

A key novelty of *CodeEvo* is our hybrid feedback mechanism, which combines deterministic compiler evaluations with generative natural language assessments to ensure the robustness and correctness of synthesized solutions. Given a transformed instruction s', the Coder first generates a candidate solution c along with initial test cases g :

$$(c, g, f_{\text{comp}}) = \text{Coder}(s') \tag{3}$$

While f_{comp} provides deterministic pass/fail signals, its utility is limited. The raw compiler output can be misleading due to insufficient test coverage and is often verbose. To overcome this, our key advantage lies in empowering the Reviewer agent to act as an intelligent judge, It moves beyond the raw signal to produce a rich, NL-based evaluation f_{NL} which scrutinizes the solution's logical alignment with the instruction, the correct implementation of specified keywords, and subtle flaws suggested by runtime behavior (*e.g.*, warnings). These two signals are then fused into a single hybrid feedback:

$$f_{\text{hybrid}} = \text{Reviewer}(f_{\text{comp}}, f_{\text{NL}}) \tag{4}$$

The hybrid feedback f_{hybrid} serves a dual role in *CodeEvo*: (1) Determining whether the current instruction-code pair should be retained or discarded. (2) Providing a rich supervisory signal that is returned to the Coder, guiding further refinement of the solution. f_{hybrid} enables iterative improvement of synthesized data without relying on external labels or human-crafted references.

3.4 Interaction-Driven Synthesis

With these mechanisms, *CodeEvo* orchestrates a collaborative refinement loop between the Coder and Reviewer. This transforms data synthesis from a static, one-shot pipeline into an adaptive process where tasks are proposed, attempted, and rigorously assessed . Crucially, the framework can dynamically adjust difficulty: if a task proves too challenging, the Reviewer can formulate a simpler Schema to generate a more tractable problem (as shown in Algorithm 1). This self-correcting loop of validation and refinement provides intrinsic quality

Algorithm 1 Single-Seed Lifecycle in *CodeEvo*.

Require: Seed instruction s, keyword set T, maximum iterations N
Ensure: Validated instruction-code pairs Q_s

Initialize $Q_s \leftarrow \varnothing$, $q \leftarrow s$, $k \leftarrow 0$
while $k < N$ **do**
  $(c, g, f_{\text{comp}}) \leftarrow \text{Coder}(q)$
  $f_{\text{NL}} \leftarrow \text{Reviewer}(q, c, g, f_{\text{comp}})$
  $f_{\text{hybrid}} \leftarrow \text{Reviewer}(f_{\text{comp}}, f_{\text{NL}})$
  if f_{hybrid} is valid **then**
    Add (q, c) to Q_s
    Sample keywords $t \subseteq T$
    Schema $\leftarrow$ Reviewer.Plan(q, t); $q^+ \leftarrow$ Reviewer.Write(q, Schema)
    $q \leftarrow q^+$
    $k \leftarrow k + 1$
  else
    Sample keywords $t' \subseteq T$
    Schema $\leftarrow$ Reviewer.Plan(q, t'); $q^- \leftarrow$ Reviewer.Write(q, Schema) ▷ Plan a simpler variant
    $(c, g, f_{\text{comp}}) \leftarrow \text{Coder}(q^-)$
    $f_{\text{NL}} \leftarrow \text{Reviewer}(q^-, c, g, f_{\text{comp}})$
    $f_{\text{hybrid}} \leftarrow \text{Reviewer}(f_{\text{comp}}, f_{\text{NL}})$
    if f_{hybrid} is valid **then**
      Add (q^-, c) to Q_s
    end if
    break
  end if
end while
return Q_s

control, which is key to maintaining a high yield of valid data. This generates a rich interaction trajectory capturing the full cycle of problem-solving, feedback, and correction, from which high-quality instruction-code pairs can be extracted.

4 Experiments

4.1 Experimental Settings

Model Settings. We evaluate our approach under two backbone agent scales: a medium-scale setting with moderately sized coder and reviewer agents, and a large-scale setting with substantially larger agent pairs. In the medium-scale configuration, we employ Qwen2.5-Coder-32B-Instruct (Hui et al., 2024) as the coder agent and Qwen2.5-32B-Instruct (Yang et al., 2024a) as the reviewer agent within the data synthesis pipelines of *CodeEvo*. For the large-scale configuration, we adopt gpt-oss-120B (OpenAI, 2025) as both the coder and reviewer agents.

To assess the performance improvements introduced by *CodeEvo* in code generation, we conduct experiments primarily on Qwen3-8B (Yang et al., 2025), representing general-purpose LLMs, and Qwen2.5-7B-Instruct (Hui et al., 2024), representing specialized CodeLLMs. Results on additional model backbones are provided in Appendix H. All inference and training is performed as full fine-tuning on interconnected clusters of 8 × A100 80GB GPUs, with more implementations details provided in Appendix A.

Evaluation Benchmarks. We evaluate the Python code generation capability of models trained on *CodeEvo* data using HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and their plus versions from EvalPlus (Liu et al., 2023). To further assess generalization under realistic difficulty levels, we also use BigCodeBench (Zhuo et al., 2025) and LiveCodeBench (Jain et al., 2025) which include more complex instructions, algorithmic logic, and function calls. Further experimental details can be found in Appendix B.

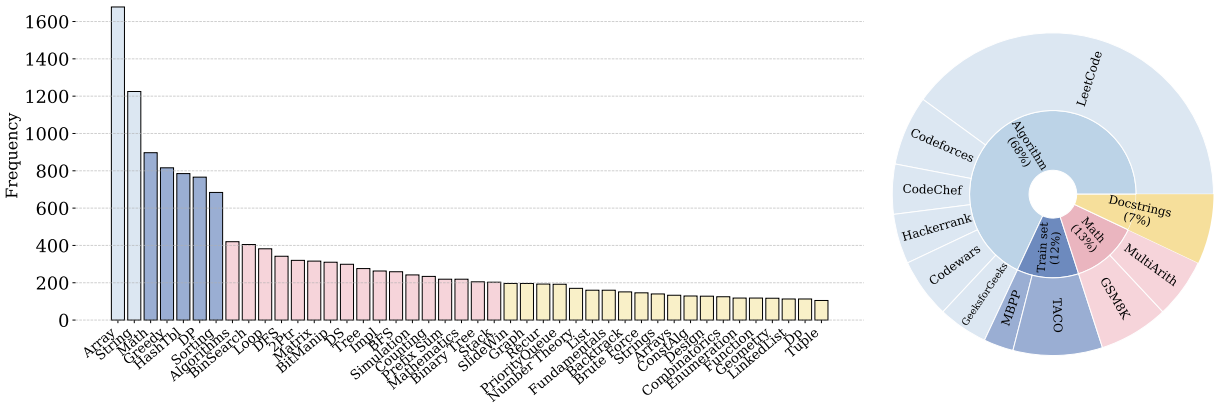
4.2 Baseline Construction

Baselines. As a pioneering study in synthesizing code-centric data, we leverage the following baselines to demonstrate the superiority of *CodeEvo*.

- **Zero-Shot**: The original evaluation setting using zero-shot prompting.
- **Code Evol-Instruct**: Proposed by WizardCoder (Luo et al., 2024b), Code Evol-Instruct first evolves task complexity and then generates the code solutions via prompting. As the original dataset is not publicly available, we reproduce this baseline under the same setting as *CodeEvo*, where Qwen2.5-32B-Instruct is used to generate instructions and Qwen2.5-Coder-32B-Instruct to generate corresponding solutions. The same seed data as *CodeEvo* is used to ensure a fair comparison.[1]
- **OSS-Instruct**: Released alongside Magicoder (Wei et al., 2024b), OSS-Instruct derives instructions from open-source code snippets written by humans and includes 75K data.

Details of the baseline construction are provided in Appendix C. All these data and resources will be

[1]We refer to this setting as `Evol-Instruct` in the following experiments for brevity.



(a) Distribution of keywords with occurrence frequency ≥ 100.

(b) Type and source of seeds.

Figure 4: An overview of seed instructions.

made public to accelerate future research.

4.3 Seed Instructions

We curate a set of ≈ 5K seed instructions from diverse sources, including programming platforms such as LeetCode and Codeforces, as well as existing math and code training sets (Cobbe et al., 2021; Austin et al., 2021). A subset of these instructions is collected alongside their corresponding reference solutions; following Luo et al. (2024b), we include them during training with appropriate ablation.

Each seed instruction is paired with a set of keyword tags (averaging 3 per instruction), which are either inherited from the original sources (*e.g.*, LeetCode tags, MBPP annotations) or assigned automatically when unavailable. Figure 4 provides an overview of the seed data used in our experiments. Additional statistics, balanced sampling strategy, and implementation details are in Appendix E.

4.4 Main Results

Performance Gains across Benchmarks. As shown in Table 1, models fine-tuned on *CodeEvo* data consistently outperform all baselines. This strong performance is evident across data synthesized from both our medium-scale (32B) and large-scale (120B) agent configurations, highlighting the framework's robustness and scalability .

Remarkably, our schema-driven method empowers even medium-sized models to produce superior data. The 17K samples generated by our 32B agents provide a greater performance boost to Qwen3-8B than the 75K OSS-Instruct dataset, further raising the LiveCodeBench score. This demonstrates that a superior synthesis algorithm can be more critical than the sheer volume of data.

Interestingly, larger gains can be observed on HE+ and MBPP+, which feature extra test cases, suggesting that our "compiler-in-the-loop" design in the hybrid feedback plays a critical role in validating functional correctness. Notably, ablation results show that removing all solutions in seeds does not lead to a notable performance drop, indicating that our grounded synthesis pipeline is robust even when fully automated.

Superior Data Efficiency. Despite using fewer training examples, *CodeEvo* outperforms other approaches, which rely on 4–5× more synthetic data and code references. It further underscores the importance of quality-aware code data construction.

The efficiency stems from innovations on both sides of the pipeline: instruction synthesis is grounded through keyword-driven refinement, while code synthesis is constrained by hybrid feedback. *CodeEvo* inherently reduces the production of invalid or redundant samples, paving the way for more data-efficient enhancement of code generation capabilities. Further discussions on the impact of data scale are presented in Section 5.3.

Ablation Studies. We perform an ablation study to isolate the effect of seed data. The results in Table 2 demonstrate that its exclusion does not lead to a significant performance degradation. In certain scenarios, models trained exclusively on *CodeEvo*-synthesized data achieve even superior performance. This further suggests that the varied and high-quality data synthesized by *CodeEvo* provides a more effective training signal than the original seed set, which is inherently more limited in its diversity and scope.

Method	Data Scale	HumanEval		MBPP		BigCodeBench-Full		BigCodeBench-Hard		LiveCodeBench
		HE	*HE+*	*MBPP*	*MBPP+*	*Instruct*	*Complete*	*Instruct*	*Complete*	*v6*
Qwen2.5-Coder-7B-Instruct	-	84.1	**79.9**	79.1	66.7	40.4	48.8	18.2	21.6	17.1
OSS-Instruct	75K	83.5	78.0	78.0	64.8	41.4	48.6	**20.3**	20.3	18.9
Coder: Qwen2.5-Coder-32B-Instruct | Reviewer: Qwen2.5-32B-Instruct										
Evol-Instruct	25K	83.5	78.0	79.1	66.9	40.6	51.4	15.5	22.3	14.5
CodeEvo	17K	85.3	79.9	81.2	68.5	41.9	52.2	17.6	26.4	22.3
Coder: GPT-OSS 120B | Reviewer: GPT-OSS 120B										
Evol-Instruct	25K	85.1	80.5	81.2	69.7	42.0	**50.3**	19.7	**23.6**	22.5
CodeEvo	17K	**86.4**	**80.9**	**83.0**	**73.2**	**43.4**	50.1	**21.5**	22.9	**24.3**
Qwen3-8B	-	82.9	77.4	80.7	70.9	42.7	49.2	14.9	22.3	39.1
OSS-Instruct	75K	84.1	78.5	79.8	67.5	42.9	50.3	15.7	24.1	36.3
Coder: Qwen2.5-Coder-32B-Instruct | Reviewer: Qwen2.5-32B-Instruct										
Evol-Instruct	25K	79.2	74.6	77.5	67.2	41.5	47.7	12.3	20.9	36.1
CodeEvo	17K	83.7	76.4	81.7	72.9	42.9	50.3	14.7	21.1	39.8
Coder: GPT-OSS-120B | Reviewer: GPT-OSS-120B										
Evol-Instruct	25K	84.5	78.2	82.4	72.5	44.1	**53.3**	16.5	**25.2**	40.9
CodeEvo	17K	**86.7**	**79.8**	**85.5**	**74.1**	**44.9**	52.5	**18.3**	24.1	**42.7**

Table 1: Results of pass@1(%) performance on various models on HumanEval(+), MBPP(+), BigCodeBench-Full, BigCodeBench-Hard, and LiveCodeBench.

Method	HE+	MBPP+	BigCodeBench-Full	
			Instruct	*Complete*
Qwen2.5-Coder-7B-Instruct				
CodeEvo w/o Seed	80.7	71.9	42.7	**51.3**
CodeEvo	**80.9**	**73.2**	**43.4**	50.1
Qwen3-8B				
CodeEvo w/o Seed	**80.1**	73.9	44.1	**52.9**
CodeEvo	79.8	**74.1**	**44.9**	52.5

Table 2: Ablation study. Use gpt-oss-120B for both coder and reviewer.

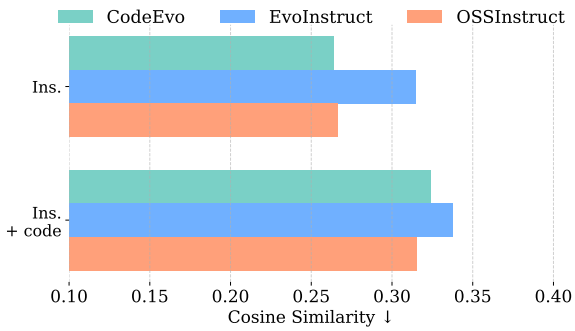


Figure 5: Comparison of instruction diversity and instruction-code pairs diversity among different synthetic methods.

5 Analysis

Beyond performance gains, we conduct a series of analyses to provide insights into the quality, diversity, scalability, and utility of synthetic code data.

5.1 Synthetic Data Diversity

A core feature of *CodeEvo* is the generation of diverse instructions and preventing overfitting to narrow problem types. To assess this, we perform a comparative diversity analysis of instruction samples and instruction-code pairs (N=1000). We compute the average pairwise cosine similarity over code embeddings [2] to assess diversity.

As shown in Figure 5, *CodeEvo* achieves the lowest average similarity among instruction samples, demonstrating the effectiveness of our keyword-guided strategy in constructing semantically diverse prompts. For instruction-code pairs, the diversity of *CodeEvo* is also comparable to OSS-Instruct, which derives data directly from human-written code. This indicates that, despite undergoing a rigid filtering process, our synthesized data retains a high level of overall diversity.

5.2 Instructions Difficulty

To evaluate whether *CodeEvo* really generates more challenging instructions, we conducted a human study comparing three variants derived from the same seed: the original seed instruction, an "evolved" version from Evol-Instruct, and the instruction synthesized by *CodeEvo*. Five participants with programming experience are invited to rate the perceived difficulty of each instruction on a scale from 1 (very easy) to 5 (very difficult).

As shown in Figure 6, *CodeEvo* instructions (from 32B settings) received the highest difficulty

[2] We leverage `text-embedding-3-small` to obtain embeddings.

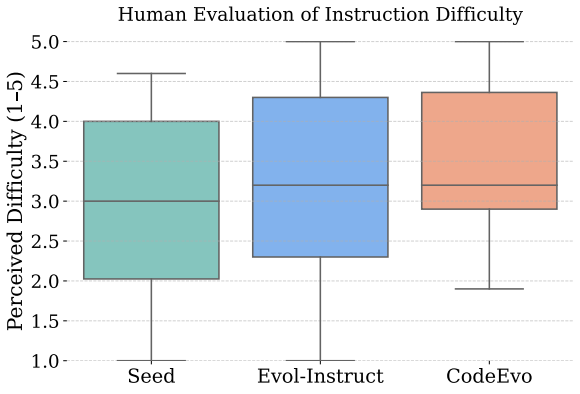


Figure 6: Human-rated difficulty of instructions.

scores (mean ≈ 3.5), followed by Evol-Instruct and Seed. In addition, *CodeEvo* exhibits a higher lower bound, indicating that the generated instructions are not only more difficult on average, but also more consistently fall within a higher difficulty range, which further validates the edge of using keyword guidance as an signal. In contrast, Evol-Instruct fails to consistently increase instruction difficulty.

5.3 Impact of Synthetic Data Scale

We investigate the scaling properties of *CodeEvo*-synthesized data with 32B scale models to back to agents. Two models are fine-tuned with incrementally larger subsets of our synthesized dataset (along with seed data), with the performance trend demonstrated in Figure 7.

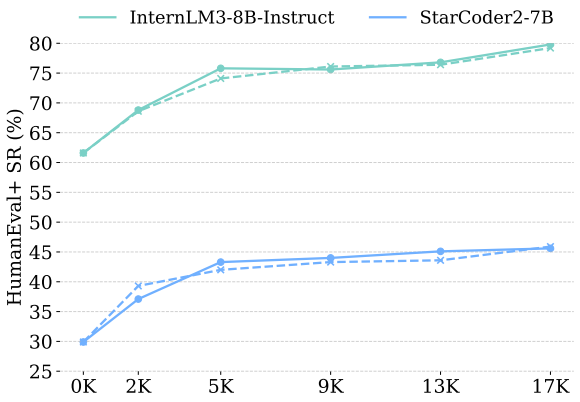


Figure 7: Impact of the scale of *CodeEvo* data. Solid lines and dashed lines indicate training with and without code references of seed data, respectively.

It can be observed that model performance improves steadily as more synthetic code is added. Importantly, this trend holds regardless of whether the training includes only syntactic data or a mixture of original and synthetic code. The results suggest that the *CodeEvo*-generated data does not introduce distributional shifts or performance degradation during scaling.

5.4 Data Survival Rate

We analyze the data survival rate, defined as the proportion of newly synthesized samples that pass both compiler checks and LLM-based evaluation. As shown in Figure 8, only a small fraction of the data is finally retained, and the survival rate steadily decreases across synthesis rounds.

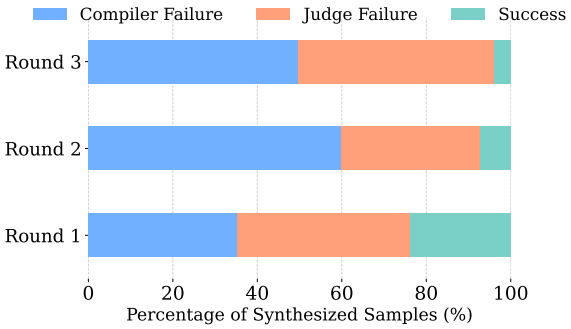


Figure 8: Synthetic data survival analysis.

This decline is both expected and desirable. First, the generated instructions become progressively more challenging, reaching the limits of the agent's capability. Second, unlike prior work that accepts instruction-code pairs after a single pass, we selectively retain only high-quality, grounded data.

6 Conclusion

In this work, we introduce *CodeEvo*, a novel framework that leverages LLM agent interactions to synthesize high-quality instruction-code pairs. To address the shortcomings of ungrounded synthetic data, *CodeEvo* employs two key mechanisms: (1) a schema-driven synthesis, where a Reviewer agent constructs a keyword-guided Schema to serve as a blueprint for a new problem's logic and complexity, and (2) a hybrid feedback loop that integrates compiler determinism with agent-based evaluation for quality control. Extensive evaluations demonstrate that models trained on *CodeEvo* data notably outperform established baselines. We find the synthesis framework's design is a critical factor, enabling even medium-sized models to generate data that surpasses larger-scale baselines. This work offers valuable insights into effective data synthesis, moving us a step closer to democratizing advanced code intelligence.

Limitations

While *CodeEvo* demonstrates the potential to overcome critical challenges in acquiring instruction-code pairs data, it is important to acknowledge certain limitations:

Dependence on Backbone Model. As a model-based approach, the quality of data synthesized by *CodeEvo* is to some extent constrained by the capabilities and biases of the backbone LLMs. While we demonstrate that the pipeline can be effectively adapted to smaller models, using stronger models is still recommended to obtain higher-quality NL queries and code solutions.

Throughput Constraints. Compared to prior prompting-based methods, *CodeEvo* introduces additional computational overhead due to multi-turn agent interactions and cross-language compilation, resulting in slower data generation throughput.

Test Case Quality. During synthesis, test cases are generated autonomously by the agent, which introduces the risk of incomplete or imperfect coverage. This is a well-known issue in software engineering, as even human-authored test suites can fail to capture all possible failure modes. Our hybrid feedback mitigates this by filtering out flawed cases, but even with this safeguard, it remains difficult to ensure complete correctness, especially at scale. We leave more robust test case generation as future work.

Broader Impacts

CodeEvo introduces a new paradigm for synthesizing high-quality code data, with the potential to improve downstream performance across a range of code generation models. Nonetheless, it is crucial to ensure that all utilized and generated code adheres to proper licensing and avoids propagating harmful coding practices.

Information About Use Of AI Assistants

In this submission, we employed LLMs to aid and polish writing, including grammar and typo checking, as well as for identifying related works.

References

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

Zhangqian Bi, Yao Wan, Zheng Wang, Hongyu Zhang, Batu Guan, Fangxin Lu, Zili Zhang, Yulei Sui, Hai Jin, and Xuanhua Shi. 2024. Iterative refinement of project-level code context for precise code generation with compiler feedback. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 2336–2353, Bangkok, Thailand. Association for Computational Linguistics.

Zheng Cai, Maosong Cao, Haojiong Chen, Kai Chen, Keyu Chen, Xin Chen, Xun Chen, Zehui Chen, Zhi Chen, Pei Chu, Xiaoyi Dong, Haodong Duan, Qi Fan, Zhaoye Fei, Yang Gao, Jiaye Ge, Chenya Gu, Yuzhe Gu, Tao Gui, Aijia Guo, Qipeng Guo, Conghui He, Yingfan Hu, Ting Huang, Tao Jiang, Penglong Jiao, Zhenjiang Jin, Zhikai Lei, Jiaxing Li, Jingwen Li, Linyang Li, Shuaibin Li, Wei Li, Yining Li, Hongwei Liu, Jiangning Liu, Jiawei Hong, Kaiwen Liu, Kuikun Liu, Xiaoran Liu, Chengqi Lv, Haijun Lv, Kai Lv, Li Ma, Runyuan Ma, Zerun Ma, Wenchang Ning, Linke Ouyang, Jiantao Qiu, Yuan Qu, Fukai Shang, Yunfan Shao, Demin Song, Zifan Song, Zhihao Sui, Peng Sun, Yu Sun, Huanze Tang, Bin Wang, Guoteng Wang, Jiaqi Wang, Jiayu Wang, Rui Wang, Yudong Wang, Ziyi Wang, Xingjian Wei, Qizhen Weng, Fan Wu, Yingtong Xiong, Chao Xu, Ruiliang Xu, Hang Yan, Yirong Yan, Xiaogui Yang, Haochen Ye, Huaiyuan Ying, Jia Yu, Jing Yu, Yuhang Zang, Chuyu Zhang, Li Zhang, Pan Zhang, Peng Zhang, Ruijie Zhang, Shuo Zhang, Songyang Zhang, Wenjian Zhang, Wenwei Zhang, Xingcheng Zhang, Xinyue Zhang, Hui Zhao, Qian Zhao, Xiaomeng Zhao, Fengzhe Zhou, Zaida Zhou, Jingming Zhuo, Yicheng Zou, Xipeng Qiu, Yu Qiao, and Dahua Lin. 2024. Internlm2 technical report. *Preprint*, arXiv:2403.17297.

Sahil Chaudhary. 2023. Code alpaca: An instruction-following llama model for code generation. https://github.com/sahil280114/codealpaca.

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

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

DeepSeek-AI, :, Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, Huazuo Gao, Kaige Gao, Wenjun Gao, Ruiqi Ge, Kang Guan, et al. 2024. Deepseek llm: Scaling open-source language models with longtermism. *arXiv preprint arXiv:2401.02954*.

Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. *SIGPLAN Not.*, 53(4):420–435.

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

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*.

Mengkang Hu, Pu Zhao, Can Xu, Qingfeng Sun, Jianguang Lou, Qingwei Lin, Ping Luo, and Saravan Rajmohan. 2025. Agentgen: Enhancing planning abilities for large language model based agent via environment and task generation. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.1*, KDD '25, page 496–507, New York, NY, USA. Association for Computing Machinery.

Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. 2024a. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *Preprint*, arXiv:2312.13010.

Siming Huang, Tianhao Cheng, Jason Klein Liu, Jiaran Hao, Liuyihan Song, Yang Xu, J. Yang, J. H. Liu, Chenchen Zhang, Linzheng Chai, Ruifeng Yuan, Zhaoxiang Zhang, Jie Fu, Qian Liu, Ge Zhang, Zili Wang, Yuan Qi, Yinghui Xu, and Wei Chu. 2024b. Opencoder: The open cookbook for top-tier code large language models.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-coder technical report. *Preprint*, arXiv:2409.12186.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. MapCoder: Multi-agent code generation for competitive problem solving. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4912–4944, Bangkok, Thailand. Association for Computational Linguistics.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2025. CodeSim: Multi-agent code generation and problem solving through simulation-driven planning and debugging. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 5113–5139, Albuquerque, New Mexico. Association for Computational Linguistics.

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

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations*.

Zaid Khan, Elias Stengel-Eskin, Jaemin Cho, and Mohit Bansal. 2025. Dataenvgym: Data generation agents in teacher environments with student feedback. In *The Thirteenth International Conference on Learning Representations*.

Shuvendu K Lahiri, Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, Madanlal Musuvathi, Piali Choudhury, Curtis von Veh, Jeevana Priya Inala, Chenglong Wang, et al. 2022. Interactive code generation via test-driven user-intent formalization. *arXiv preprint arXiv:2208.05950*.

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023. CAMEL: Communicative agents for "mind" exploration of large language model society. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. 2023. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue

Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2024. Starcoder 2 and the stack v2: The next generation. *Preprint*, arXiv:2402.19173.

Haipeng Luo, Qingfeng Sun, Can Xu, Pu Zhao, Qingwei Lin, Jianguang Lou, Shifeng Chen, Yansong Tang, and Weizhu Chen. 2024a. Arena learning: Build data flywheel for llms post-training via simulated chatbot arena. *Preprint*, arXiv:2407.10627.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2024b. Wizardcoder: Empowering code large language models with evol-instruct. In *The Twelfth International Conference on Learning Representations*.

Wei Ma, Shangqing Liu, Zhihao Lin, Wenhan Wang, Qiang Hu, Ye Liu, Cen Zhang, Liming Nie, Li Li, and Yang Liu. 2024. Lms: Understanding code syntax and semantics for code analysis. *Preprint*, arXiv:2305.12138.

Yichuan Ma, Yunfan Shao, Peiji Li, Demin Song, Qipeng Guo, Linyang Li, Xipeng Qiu, and Kai Chen. 2025. Unitcoder: Scalable iterative code synthesis with unit test guidance. *Preprint*, arXiv:2502.11460.

Somshubra Majumdar, Vahid Noroozi, Mehrzad Samadi, Sean Narenthiran, Aleksander Ficek, Wasi Uddin Ahmad, Jocelyn Huang, Jagadeesh Balam, and Boris Ginsburg. 2025. Genetic instruct: Scaling up synthetic generation of coding instructions for large language models. *Preprint*, arXiv:2407.21077.

Arindam Mitra, Luciano Del Corro, Guoqing Zheng, Shweti Mahajan, Dany Rouhana, Andres Codas, Yadong Lu, Wei ge Chen, Olga Vrousgos, Corby Rosset, Fillipe Silva, Hamed Khanpour, Yash Lara, and Ahmed Awadallah. 2024. Agentinstruct: Toward generative teaching with agentic flows. *Preprint*, arXiv:2407.03502.

OpenAI. 2025. gpt-oss-120b & gpt-oss-20b model card. *gpt-oss model card*, 1:1.

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. *Preprint*, arXiv:2304.03442.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. ChatDev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15174–15186, Bangkok, Thailand. Association for Computational Linguistics.

Yunfan Shao, Linyang Li, Yichuan Ma, Peiji Li, Demin Song, Qinyuan Cheng, Shimin Li, Xiaonan Li, Pengyu Wang, Qipeng Guo, Hang Yan, Xipeng Qiu, Xuanjing Huang, and Dahua Lin. 2025. Case2Code: Scalable synthetic data for code generation. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 11056–11069, Abu Dhabi, UAE. Association for Computational Linguistics.

Theodore Sumers, Shunyu Yao, Karthik Narasimhan, and Thomas Griffiths. 2024. Cognitive architectures for language agents. *Transactions on Machine Learning Research*. Survey Certification.

Qiushi Sun, Zhirui Chen, Fangzhi Xu, Kanzhi Cheng, Chang Ma, Zhangyue Yin, Jianing Wang, Chengcheng Han, Renyu Zhu, Shuai Yuan, et al. 2024a. A survey of neural code intelligence: Paradigms, advances and beyond. *arXiv preprint arXiv:2403.14734*.

Qiushi Sun, Kanzhi Cheng, Zichen Ding, Chuanyang Jin, Yian Wang, Fangzhi Xu, Zhenyu Wu, Chengyou Jia, Liheng Chen, Zhoumianze Liu, et al. 2024b. Os-genesis: Automating gui agent trajectory construction via reverse task synthesis. *arXiv preprint arXiv:2412.19723*.

Qiushi Sun, Zhangyue Yin, Xiang Li, Zhiyong Wu, Xipeng Qiu, and Lingpeng Kong. 2023. Corex: Pushing the boundaries of complex reasoning through multi-model collaboration. *arXiv preprint arXiv:2310.00280*.

Xin Wang, Yasheng Wang, Yao Wan, Fei Mi, Yitong Li, Pingyi Zhou, Jin Liu, Hao Wu, Xin Jiang, and Qun Liu. 2022. Compilable neural code generation with compiler feedback. In *Findings of the Association for Computational Linguistics: ACL 2022*, pages 9–19, Dublin, Ireland. Association for Computational Linguistics.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. 2025a. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*.

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Program synthesis using abstraction refinement. *Proc. ACM Program. Lang.*, 2(POPL).

Yaoxiang Wang, Haoling Li, Xin Zhang, Jie Wu, Xiao Liu, Wenxiang Hu, Zhongxin Guo, Yangyu Huang, Ying Xin, Yujiu Yang, Jinsong Su, Qi Chen, and Scarlett Li. 2025b. Epicoder: Encompassing diversity and complexity in code generation. *Preprint*, arXiv:2501.04694.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. 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)*, pages 13484–13508, Toronto, Canada. Association for Computational Linguistics.

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

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024b. Magicoder: Empowering code generation with OSS-instruct. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 52632–52657. PMLR.

XTuner Contributors. 2023. Xtuner: A toolkit for efficiently fine-tuning llm. `https://github.com/InternLM/xtuner`.

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

Fangzhi Xu, Qiushi Sun, Kanzhi Cheng, Jun Liu, Yu Qiao, and Zhiyong Wu. 2024b. Interactive evolution: A neural-symbolic self-training framework for large language models. *arXiv preprint arXiv:2406.11736*.

Fangzhi Xu, Zhiyong Wu, Qiushi Sun, Siyu Ren, Fei Yuan, Shuai Yuan, Qika Lin, Yu Qiao, and Jun Liu. 2024c. Symbol-LLM: Towards foundational symbol-centric interface for large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13091–13116, Bangkok, Thailand. Association for Computational Linguistics.

Fangzhi Xu, Hang Yan, Chang Ma, Haiteng Zhao, Qiushi Sun, Kanzhi Cheng, Junxian He, Jun Liu, and Zhiyong Wu. 2025a. Genius: A generalizable and purely unsupervised self-training framework for advanced reasoning. *arXiv preprint arXiv:2504.08672*.

Zhangchen Xu, Fengqing Jiang, Luyao Niu, Yuntian Deng, Radha Poovendran, Yejin Choi, and Bill Yuchen Lin. 2025b. Magpie: Alignment data synthesis from scratch by prompting aligned LLMs with nothing. In *The Thirteenth International Conference on Learning Representations*.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. 2024a. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*.

Zonghan Yang, Peng Li, Ming Yan, Ji Zhang, Fei Huang, and Yang Liu. 2024b. React meets actre: Autonomous annotation of agent trajectories for contrastive self-training. In *First Conference on Language Modeling*.

Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiufeng Yin. 2024. WaveCoder: Widespread and versatile enhancement for code large language models by instruction tuning. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5140–5153, Bangkok, Thailand. Association for Computational Linguistics.

Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Wang Yongji, and Jian-Guang Lou. 2023. Large language models meet NL2Code: A survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7443–7464, Toronto, Canada. Association for Computational Linguistics.

Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhu Chen. 2025. Acecoder: Ac-

ing coder rl via automated test-case synthesis. *ArXiv*, 2502.01718.

Kechi Zhang, Ge Li, Yihong Dong, Jingjing Xu, Jun Zhang, Jing Su, Yongfei Liu, and Zhi Jin. 2024. Codedpo: Aligning code models with self generated and verified source code. *Preprint*, arXiv:2410.05605.

Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. 2024a. OpenCodeInterpreter: Integrating code generation with execution and refinement. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 12834–12859, Bangkok, Thailand. Association for Computational Linguistics.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. 2024b. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Bangkok, Thailand. Association for Computational Linguistics.

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

A Model and Training Details

To validate that our method can generalize to different LLM architectures and paradigms, we experimented with four widely used base models.

InternLM3-8B-Instruct InternLM3-8B-Instruct is a typical general-purpose large language model. It follows the architecture of its predecessor models (Cai et al., 2024) and is trained on 4 trillion high-quality tokens to support superior capabilities in multiple domains. It also supports long context understanding and CoT reasoning. In our experiment, we use this model to validate that our pipeline can improve the coding performance of general-purpose instruction-tuned models.

Qwen2.5-Coder-7B-Instruct Qwen2.5-Coder-7B-Instruct (Hui et al., 2024) is an instruction-tuned language model specifically enhanced for coding tasks. It builds upon the architecture of its general-purpose base model, Qwen2.5 (Yang et al., 2024a), inheriting its computational efficiency and versatile vocabulary. To ensure the integrity of code understanding and generation, the model also incorporates several special tokens explicitly designed for code block generation. The model is trained on over 18 trillion tokens and incorporated extensive post-training technique, making it an ideal test bed for us to evaluate our method on coding LLMs that is transformed from general LLMs.

DeepSeek-Coder-6.7B-Instruct DeepSeek-Coder-6.7B-Instruct (Guo et al., 2024) is an instruction-tuned codeLLM based on the architecture of the Deepseek model (DeepSeek-AI et al., 2024). It is trained on a corpus of 2 trillion tokens, extracted through a meticulously designed pipeline tailored for coding data. Compared to general-purpose LLMs, it employs a relatively small vocabulary specifically optimized for code-related tasks. In our experiments, we adopot this model as a representative domain-specific LLM to evaluate the effectiveness of our method on coding-oriented models.

StarCoder2-7B StarCoder2-7B (Lozhkov et al., 2024) is a representative base model from the early era of codeLLMs. It is pretrained on 3.5 trillion tokens without any additional post-training. Similar to DeepSeek-Coder, StarCoder2 employs a customized vocabulary for code-related task. We evaluate our method on this model to assess whether

our trajectory data remains effective in the absence of human alignment.

For instruction-tuned models, we adopt XTuner framework (XTuner Contributors, 2023) to streamline training. For StarCoder2-7B, we employ LLaMA-Factory (Zheng et al., 2024b) to conduct supervised fine-tuning. Following previous practice and our observations, we use a learning rate of 2×10^{-6} for more stable training. For the rest of models, we used a learning rate of 5×10^{-6}.

All of our models are trained on 8 × NVIDIA H800 GPUs, with a batch size of 4 per device, and a gradient accumulation of 2 steps.

B Evaluation Details

HumanEval & MBPP. HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) are two common code completion benchmarks for evaluating the coding capability of LLMs. To further extend these two datasets, EvalPlus (Liu et al., 2023) introduced HumanEval+ and MBPP+ by adding more challenging test cases and correcting inaccurate solutions. In this study, we used both the original benchmarks (HumanEval and MBPP) and their augmented versions (HumanEval+ and MBPP+) to evaluate models trained on our data as well as baseline models. We employed the official EvalPlus implementation to evaluation both benchmarks and reported 0-shot results for all variants. Release under MIT License.

BigCodeBench. BigCodeBench (Zhuo et al., 2025) is a challenging benchmark for code generation, aimed at evaluating models' ability to interpret complex instructions and invoke diverse external libraries correctly. Under the completion setting, each task provides a function signature and docstrings, requiring the model to generate the full function implementation. Under the instruction setting, models are required to generate corresponding code according to a given instruction. A unit test is also provided to verify functional correctness. Spanning a broad range of practical programming scenarios, BigCodeBench assesses models on real-world tasks that demand precise understanding of task-specific APIs and library usage. It is released under the Apache License 2.0.

LiveCodeBench. LiveCodeBench (Jain et al., 2024) is a comprehensive coding benchmark curated from mainstream competition programming platforms. It aims to provide an up-to-date, contamination-free evaluating testbed, and is continuously updated with new versions that aggregate additional problems over time. In our experiments, we use the *release_v6* version of the dataset, which comprises 1055 problems collected between May 2023 and Apr 2025.

C Details of Baselines

C.1 Evol Instruct

We follow the Evol-Instruct baseline implementation used in WizardCoder (Luo et al., 2024b). To ensure a fair comparison with *CodeEvo*, we reproduce this baseline under the same experimental setup: Qwen2.5-32B-Instruct is used to generate instructions, and Qwen2.5-Coder-32B-Instruct is used to synthesize the corresponding code solutions. We adopt the same prompt heuristics as in the original implementation, where each seed is expected to produce five instruction–code pairs. The same set of seed data as used in *CodeEvo* is employed, and the seed instructions are included in the fine-tuning process.

C.2 OSS-Instruct

We leverage OSS-Instruct (Wei et al., 2024b) from Magicoder as another strong baseline. Specifically, we directly use the full 75K dataset released by the authors and perform fine-tuning under the same hyperparameter settings as used for *CodeEvo*.

D Code Synthesis Details

The prompts we used for agent collaboration are in Prompt 9.

E Seed Instructions and Keywords

We collect instruction data from a variety of public coding platforms, including

- LeetCode: `https://leetcode.com/`
- Codeforces: `https://codeforces.com/`
- Codewars: `https://www.codewars.com/`
- GeeksforGeeks: `https://www.geeksforgeeks.org/`
- CodeChef: `https://www.codechef.com/`

We conduct a thorough similarity check and confirm that there is no contamination with the evaluation benchmarks. No personally identifiable information is present in the dataset.

The prompt used for keyword generation is provided in Prompt 10, and the keyword sampling algorithm is detailed in Algorithm 2.

Prompt for Generating CodeEvo Trajectory

Coder:
Write python code to solve the following problem:
{Problem description}.
Include test case execution in your code.

Reviewer:
Next I will give you a coding problem, a piece of code, and the execution result of this code. Please determine if the code given correctly solves the problem given.
The problem is described as:
{Problem description}
The code to be assessed is:
{Code from Coder}
The output of this code during execution is:
{Outputs from execution}
The error message generated during execution is:
{Errors from execution}
First output S̈uccessör F̈ailureäs your judgement. Then explain the reasons and possible improvements. Do not give out improved codes.

Coder:
The following is an evaluation and feedback on whether the code you generated successfully answered the given question:
{Feedbacks from Reviewer}
Please use this feedback to improve your code so that it answers the question correctly. Still, output the refined code block only.

Reviewer:
Below I will give you a programming problem and its keywords, design a programming problem based on this programming problem that is knowledge related but more difficult.
You can increase the difficulty by using, but not limited to, the following methods:
{Approaches to increase problem difficulty}
Please use the following output format:
###New
New programming problem you designed
This original programming problem is described as:
{Problem description}
The keywords of the original problem are:
{Keywords of the problem}

Prompt 9: Prompts for generating CodeEvo Trajectory.

Algorithm 2 Stratified Keyword Sampling Algorithm

Require: Label set T, $m = |T|$; sampling range $[r_{\min}, r_{\max}]$; maximum sampling steps $t_{\max}$

if $m \leq r_{\max}$ **then**
  for $t = 1$ to $t_{\max}$ **do**
    Randomly sample $r \sim \mathcal{U}[r_{\min}, \min(m, r_{\max})]$
    Sample a subset $S_t \subseteq T$, where $|S_t| = r$
  end for
else
  Initialize $T_{\text{remaining}} \leftarrow T$
  for $t = 1$ to $t_{\max}$ **do**
    if $T_{\text{remaining}} = \varnothing$ **then**
      break
    end if
    Let $r \sim \mathcal{U}[r_{\min}, \min(|T_{\text{remaining}}|, r_{\max})]$
    Sample $S_t \subseteq T_{\text{remaining}},\ |S_t| = r$
    $T_{\text{remaining}} \leftarrow T_{\text{remaining}} \setminus S_t$
  end for
end if

```
Prompt for Generating Keywords for Seed Instructions
You are given a text that includes a programming problem description and explanations of its solutions.
Your task is to identify and list the key programming concepts, data structures, or algorithms that
are central to solving the problem. Provide your answer as a list of keywords or tags (e.g., "Array",
"Hash Table", "Sorting", "Recursion", "Loop", "String", "Stack") that best capture the main ideas or
techniques involved.
For example, if the problem involves finding two numbers in an array that add up to a target sum,
appropriate tags might be "Array" and "Hash Table".
Now, here is the text:
{text}
Please provide the keywords for this problem as a comma-separated list (e.g., "Array, Hash Table").
```

Prompt 10: Prompts for generating keywords for instructions.

F Details of Ablation Studies

In addition to Table 2, we provide the complete results of ablation studies in Table 3 that covers more benchmarks.

G Extended Analysis

G.1 Impact of Model Scale

Beyond the MBPP+ experiments, we further demonstrate the scale of agent backbones' influence on downstream results, as shown in Figure 11.

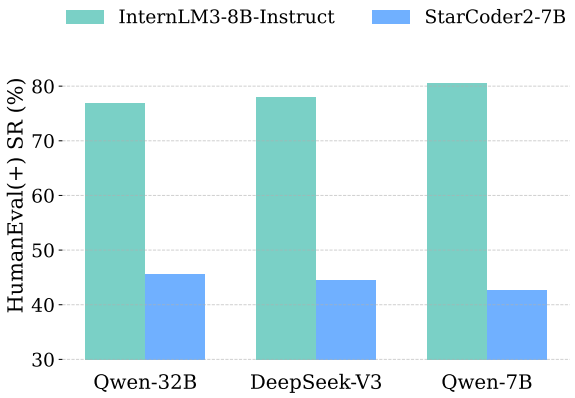


Figure 11: Success rates of InternLM3 and StarCoder2 after training with data synthesized through agents with different backbones.

G.2 Solvable Rate of Synthetic Instructions

As discussed, a key pitfall of synthetic code data is that newly generated instructions may be ungrounded, *i.e.*, cannot find valid solutions. We investigate this issue by conducting a manual analysis of instructions synthesized by *CodeEvo* and Evol-Instruct.

The results shown in Figure 12 reveal clear differences in solvability across the two approaches.

Figure 12: Comparing the solvability of instructions synthesized by *CodeEvo* and Evol-Instruct.

H Results on Additional Backbones

To further validate the effectiveness of our method, we conduct the main experiment on 3 additional backbones: InternLM3-8B-Instruct (Cai et al., 2024), StarCoder2-7B (Lozhkov et al., 2024) and DeepSeek-Coder-6.7B-Instruct (Guo et al., 2024). We used the medium-scale agent configuration (Qwen2.5-Coder-32B-Instruct + Qwen2.5-32B-Instruct) in this additional experiment. The results are shown in Table 4. We can easily see that our method outperforms most of the baselines on all backbones.

I Human Participants

We recruit college-level participants with a background in computer science to conduct experiments in Section 5.2 and Appendix G.2. For instructions, participants are asked to follow the synthesized instructions directly as part of the evaluation process.

All participants are compensated at a rate of $10 per hour for their time and effort. We do not record any personal information, and all participants pro-

Method	Data Scale	HumanEval		MBPP		BigCodeBench-Full		BigCodeBench-Hard	
		HE	*HE+*	*MBPP*	*MBPP+*	*Instruct*	*Complete*	*Instruct*	*Complete*
InternLM3-8B-Instruct									
CodeEvo w/o Seed	12K	80.5	**76.8**	**81.5**	**71.4**	**34.9**	41.3	14.86	**16.20**
CodeEvo	17K	**82.3**	**76.8**	81.2	**71.4**	**34.9**	**43.2**	**15.54**	15.50
StarCoder2-7B									
CodeEvo w/o Seed	12K	**51.2**	**46.3**	64.0	52.9	29.7	33.8	**8.8**	6.8
CodeEvo	17K	50.0	44.5	**66.4**	**55.6**	**30.3**	**34.6**	**8.8**	**10.8**
DeepSeek-Coder-6.7B-Instruct									
CodeEvo w/o Seed	12K	76.2	68.9	**77.2**	**65.9**	36.0	43.4	**12.8**	**18.6**
CodeEvo	17K	**77.4**	**71.3**	**77.2**	**65.9**	**37.5**	43.4	**12.8**	16.9
Qwen2.5-Coder-7B-Instruct									
CodeEvo w/o Seed	12K	84.8	79.3	78.0	64.8	**42.0**	52.1	17.6	23.6
CodeEvo	17K	**85.3**	**79.9**	**81.2**	**68.5**	41.9	**52.2**	17.6	**26.4**

Table 3: Ablation study comparing *CodeEvo* with and without Seed Initialization across multiple backbones and benchmarks, while maintaining consistent data scale annotation.

Method	Data Scale	HumanEval		MBPP		BigCodeBench-Full		BigCodeBench-Hard		LiveCodeBench
		HE	*HE+*	*MBPP*	*MBPP+*	*Instruct*	*Complete*	*Instruct*	*Complete*	*v6*
InternLM3-8B-Instruct	-	64.0	61.6	64.8	54.5	26.4	41.3	10.14	12.20	16.0
Evol-Instruct	25K	68.3	64.6	72.4	62.7	31.5	39.5	12.84	14.90	14.9
OSS-Instruct	75K	80.5	71.4	80.4	70.4	30.1	40.2	14.19	14.90	15.4
CodeEvo	17K	**82.3**	**78.0**	**81.2**	**71.4**	**34.9**	**43.2**	**15.54**	**15.50**	**17.1**
StarCoder2-7B	-	35.4	29.9	54.4	45.6	8.8	10.7	0.6	4.1	0.6
Evol-Instruct	25K	45.7	42.7	60.6	51.3	29.2	33.2	8.1	6.1	10.9
OSS-Instruct	75K	**50.6**	43.9	60.3	49.7	29.7	31.4	7.4	6.8	**12.6**
CodeEvo	17K	50.0	**44.5**	**66.4**	**55.6**	**30.3**	**34.6**	**8.8**	**10.8**	**12.6**
DeepSeek-Coder-6.7B-Instruct	-	74.4	68.9	74.3	65.6	34.6	43.4	9.5	16.9	14.3
Evol-Instruct	25K	75.0	68.3	74.9	64.6	35.8	**44.6**	**12.8**	16.9	13.1
OSS-Instruct	75K	76.8	70.7	**77.2**	64.6	36.4	43.8	12.2	11.5	14.9
CodeEvo	17K	**77.4**	**71.3**	**77.2**	**65.9**	**37.5**	43.8	**12.8**	**18.2**	**17.7**

Table 4: Extended results on additional backbone models (directly using keywords). All results are reported with pass@1(%) performance.

vide informed consent. The experiment does not involve surveys, interviews, or behavioral tracking.

J Case Studies

We provide two case studies of *CodeEvo* generating new instructions, as shown in Prompt 14 and Prompt 15.

Instructions for Human Difficulty Evaluation
You are provided with a set of three programming instructions derived from the same problem: (1) some coding problems from the internet, (2) some coding problems written by LLMs, and (3) some coding problems generated by a new framework.
Your task is to evaluate the perceived difficulty of each instruction independently, based on your intuition as a programmer. Difficulty refers to how challenging you find the task described in the instruction, considering both problem complexity and clarity.
Please rate each instruction on a scale from 1 (very easy) to 5 (very difficult).
You do not need to write code or solve the problem—just focus on how difficult it seems to you!
Rating scale:
1 = Very easy (e.g., straightforward and simple logic)
2 = Easy (e.g., minor reasoning or implementation effort)
3 = Moderate difficulty (e.g., standard algorithm, some complexity)
4 = Hard (e.g., non-trivial logic or multi-step reasoning)
5 = Very difficult (e.g., involves advanced reasoning and multiple functions)

Prompt 13: Instructions given to human annotators for difficulty evaluation.

Case of building programming problems
Seed Problem:
A permutation of an array of integers is an arrangement of its members into a sequence or linear order. For example, for arr = [1,2,3], the following are all the permutations of arr: [1,2,3], [1,3,2], [2, 1, 3], [2, 3, 1], [3,1,2], [3,2,1].
The next permutation of an array of integers is the next lexicographically greater permutation of its integer. More formally, if all the permutations of the array are sorted in one container according to their lexicographical order, then the next permutation of that array is the permutation that follows it in the sorted container. If such arrangement is not possible, the array must be rearranged as the lowest possible order (i.e., sorted in ascending order).
For example, the next permutation of arr = [1,2,3] is [1,3,2].
Similarly, the next permutation of arr = [2,3,1] is [3,1,2].
While the next permutation of arr = [3,2,1] is [1,2,3] because [3,2,1] does not have a lexicographical larger rearrangement.
Given an array of integers nums, find the next permutation of nums.
The replacement must be in place and use only constant extra memory.
Generated Problem 1:
Given an array of integers ‘nums‘, find the next lexicographically greater permutation of its integers. If such an arrangement is not possible, rearrange the array in ascending order. Additionally, ensure that the solution handles arrays with duplicate elements efficiently and outputs the result in a circular format, where the last element points back to the first element.
Generated Problem 2:
Given a doubly linked list of integers ‘nums‘, find the next lexicographically greater permutation of its integers. If such an arrangement is not possible, rearrange the list in ascending order. The solution must handle lists with duplicate elements efficiently and output the result in a circular format, where the last node points back to the first node and vice versa. Additionally, the solution should be able to handle lists with up to 1000 nodes, where each node’s value is between 0 and 1000.

Prompt 14: Case Study.

Case of building programming problems
Seed Problem:
You are given a string S of length 6. It is guaranteed that the first three characters of S are ABC and the last three characters are digits.
Determine if S is the abbreviation of a contest held and concluded on AtCoder before the start of this contest.
Here, a string T is ẗhe abbreviation of a contest held and concluded on AtCoder before the start of this contestïf and only if it equals one of the following 348 strings:
ABC001, ABC002, . . . , ABC314, ABC315, ABC317, ABC318, . . . , ABC348, ABC349.
Note that ABC316 is not included.
Input
The input is given from Standard Input in the following format:
S
Output
If S is the abbreviation of a contest held and concluded on AtCoder before the start of this contest, print Yes; otherwise, print No.
Constraints
- S is a string of length 6 where the first three characters are ABC and the last three characters are digits.
Generated Problem 1:
You are given a string S of length 6. It is guaranteed that the first three characters of S are ABC and the last three characters are digits. Determine if S is the abbreviation of a contest held and concluded on AtCoder before the start of this contest, considering that some contest numbers were skipped due to special events. Specifically, the valid contest numbers are ABC001 through ABC314, ABC315, ABC317 through ABC348, and ABC349. Note that ABC316 is not included. Additionally, validate the input to ensure it strictly follows the format and handle any invalid inputs gracefully.
Generated Problem 2:
You are given a string S of length 6. It is guaranteed that the first three characters of S are ABC and the last three characters are digits. Determine if S is the abbreviation of a contest held and concluded on AtCoder before the start of this contest, considering that some contest numbers were skipped due to special events. Specifically, the valid contest numbers are ABC001 through ABC314, ABC315, ABC317 through ABC348, and ABC349. Note that ABC316 is not included. Additionally, validate the input to ensure it strictly follows the format and handle any invalid inputs gracefully. Furthermore, if the input is valid but the contest number is a multiple of 7, print S̈pecial Caseïnstead of Ÿes.
Generated Problem 3:
You are given a string S of length 6. It is guaranteed that the first three characters of S are ABC and the last three characters are digits. Determine if S is the abbreviation of a contest held and concluded on AtCoder before the start of this contest, considering that some contest numbers are missing due to special events. Specifically, the valid contest numbers are all integers from 001 to 349 except for 316. Additionally, validate the input to ensure it strictly follows the format and handle any invalid inputs gracefully by printing Ïnvalid Input.

Prompt 15: Case Study.