UnitCoder: Scalable Code Synthesis from Pre-training Corpora

Yichuan Ma[1,2], Yunfan Shao[1,2], Peiji Li [1,2], Demin Song[2], Qipeng Guo[2], Linyang Li[2†], Xipeng Qiu[1], Kai Chen[2],
[1]School of Computer Science, Fudan University, Shanghai,
[2]Shanghai AI Laboratory, Shanghai,
yichuanma24@m.fudan.edu.cn,lilinyang@pjlab.org.cn

Abstract

Large Language Models (LLMs) have demonstrated remarkable capabilities in various tasks, yet code generation remains a major challenge. Despite the abundant sources of code data, constructing high-quality training datasets at scale poses a significant challenge. Pre-training code data typically suffers from inconsistent data quality issues. Conversely, instruction-based methods which use a high-quality subset as seed samples suffer from limited task diversity. In this paper, we introduce **UnitCoder**, which directly supervises pre-training data quality through automatically generated unit tests, while ensuring the correctness via an iterative fix and refine flow. Code synthesized by UnitCoder benefits from both the diversity of pre-training corpora and the high quality ensured by unit test supervision. Our experiments demonstrate that models fine-tuned on our synthetic dataset exhibit consistent performance improvements. Our work presents a scalable approach that leverages model-generated unit tests to guide the synthesis of high-quality code data from pre-training corpora, demonstrating the potential for producing diverse and high-quality post-training data at scale. All code and data will be released[1].

1 Introduction

Large language models (LLMs) have demonstrated remarkable capabilities in coding tasks, as evidenced by both general LLMs and code-specilaized models. Leading foundation models like OpenAI o1[2], GPT-4o (Achiam et al., 2023), Claude[3], and DeepSeek R1 (DeepSeek-AI et al., 2025) excel at code understanding and generation. Meanwhile, specialized models such as CodeLlama (Rozière et al., 2023), Deepseek-Coder (Guo et al., 2024) and Qwen-Coder (Hui et al., 2024) have emerged as powerful coding assistants.

Despite the success of LLMs in coding tasks, it still remains a challenge acquiring high-quality code data. Pre-training corpora typically suffer from inconsistent data quality. To address this issue, mainstream code-specialized models often combine large-scale code pre-training data with high-quality instruction data to ensure the accuracy of model-generated code (Guo et al., 2024; Yang et al., 2024b; Hui et al., 2024). In the domain of synthetic data generation, prevailing approaches focus on using high-quality instruction data as seeds and employing prompt-based methods to synthesize instructional data. While these methods ensure the quality of synthetic data, their inherent diversity is inevitably constrained by the limitations of the original seed data (Luo et al., 2023; Huang et al., 2023; Yu et al., 2023).

Therefore, a natural idea is to introduce direct supervision for pre-training data to leverage the diversity of the original pre-training code corpus. Several works have already considered similar approaches. For instance, OSS-Instruct (Wei et al., 2024c) and Code-DPO (Zhang et al., 2024) explored methods to first synthesize instructions based on pre-training data, and then apply quality supervision using model-generated test cases.

Inspired by these approaches, we explore the possibility of models learning directly from pre-training corpora without synthesized instructions. We contend that using powerful LLMs to generate instructions introduces biases that may limit the diversity inherent in the original code corpus. Moreover, this instruction synthesis process may lead to further computational costs. We propose that directly supervising the quality of pre-training code while preserving its original functionality represents a more straightforward and natural approach.

Based on this motivation, we propose **UnitCoder**, a Unit Test-based code synthesis frame-

[†]Corresponding Author.
[1]https://github.com/Entarochuan/UnitCoder
[2]https://openai.com/o1/
[3]https://www.anthropic.com/claude

Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, pages 5624–5642
November 4-9, 2025

work. Specifically, we supervise the original pre-training code corpus using generated unit tests. For code snippets that fail to pass the unit tests, we implement an iterative fix and refine flow to perform multiple rounds of corrections to the code details, adjusting code correctness without modifying the original functionality of the code snippets. To comprehensively evaluate both the diversity and accuracy of data synthesized by the UnitCoder method, we select the API call scenario as our code application context and adapt BigCodeBench (Zhuo et al., 2024) as the primary evaluation benchmark.

The UnitCoder framework consists of three key components: (i) Data Preparation, (ii) Fix and Refine Flow and (iii) Post-Train. First, we utilize the AST (Abstract Syntax Tree) parsing tool[4] to extract syntactically valid code snippets from the pre-training corpora. Additionally, we develop a unit test generator fine-tuned on human-written Python test cases, capable of validating complex API calls and edge cases. In the second stage, for functions that fail the unit tests, we employ a bug-fix agent to iteratively debug and modify code snippets based on failure traces. Once the code passes the unit test, we introduce a refine agent to improve code style and readability without altering functionality. In the final stage, we conduct post-training on base models. All agents in our experiment are implemented using open-source LLMs, including Llama3-70B and Qwen2.5-72B. Utilizing UnitCoder, we successfully synthesize over 500K executable code data, covering over 370 unique API calls.

We evaluate our approach through by fine-tuning the Llama (Dubey et al., 2024) and InternLM (Cai et al., 2024) series models with UnitCoder-synthesized data. Results demonstrate that the post-training stage improves model performance across all coding benchmarks, with the most noticeable improvement seen on BigCodeBench, where complex API interactions are required. Our contributions can be summarized as follows:

- We present UnitCoder, a scalable framework for synthesizing high-quality post-training code data from raw code corpora under unit test guidance. UnitCoder ensures the synthesis of high-quality data while preserving the original code functionality.
- We generate a dataset of 500K+ verifiable programs using UnitCoder. Experiments demonstrate that our synthetic data consistently improves base models' performance on code generation benchmarks, particularly in handling complex API interactions.
- We conduct ablation studies to validate each component's necessity and analyze the relationships between data scale, diversity, and model performance, providing insights for scalable code synthesis.

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

2 Related Work

Code LLMs Code LLM developments have progressed along two main directions: large-scale pre-training and specialized instruct-tuning. Early works of code pre-training models include pioneering works like CodeX (Chen et al., 2021b), CodeGen (Nijkamp et al., 2023), StarCoder (Li et al., 2023) and CodeLlama (Rozière et al., 2023). Open-sourcing series such as Qwen (Bai et al., 2023; Yang et al., 2024a) and Deepseek (DeepSeek-AI et al., 2024) proposed specialized code models as well, examplified by Qwen-Coder and Deepseek-Coder(Hui et al., 2024; Guo et al., 2024).

LLM-Based Code Filtering and Generation Leveraging LLMs for code synthesis is a effective approach. First, LLM-based data filtering provides a valuable quality supervision signal in data preparation. For example, WaveCoder (Yu et al., 2023) employed GPT-4 as a discriminator, while Arctic-SnowCoder (Wei et al., 2024b) explored the potential of BERT-based models for code data filtering.

In parallel, works represented by WizardCoder (Luo et al., 2023) focused on enhancing instruction diversity through improved instruction engineering with powerful LLMs (Jiang et al., 2024; Zan et al., 2023; Zhu et al., 2022). Additionally, research exemplified by AgentCoder (Huang et al., 2023) investigated prompt-based approaches that integrate test cases and multi-agent collaboration to improve coding performance (Huang et al., 2023; Chen et al., 2022; Islam et al., 2024). Methods like those used in WarriorCoder (Feng et al., 2024) also construct well-designed multi-agent frameworks for synthesizing code data.

LLM-Based Unit Test Generation Meanwhile, using LLM-generated test cases or unit tests to supervise code quality is becoming a research hotspot. Initially, several works explored specific strategies

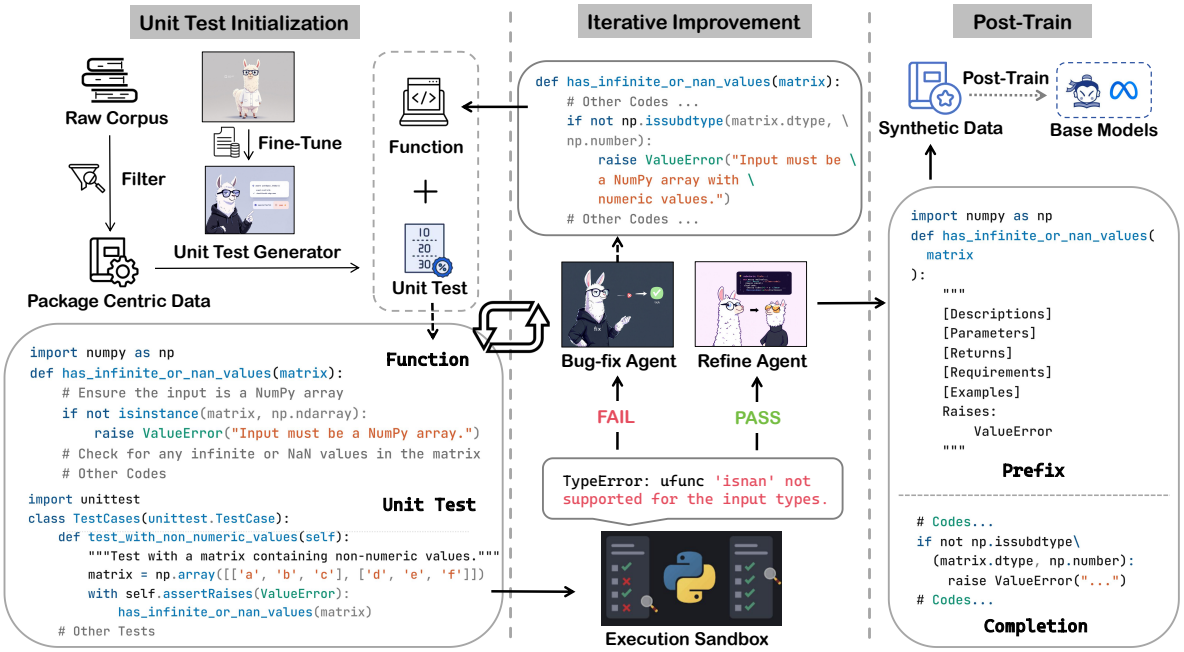

Figure 1: **The UnitCoder pipeline.** The pipeline consists of three main stages: (1) Data Preparation - filter package-centric data from raw code corpus and fine-tune a unit test generator to produce corresponding tests; (2) Fix and Refine Flow - execute function-test pairs in sandbox, iteratively fix failed cases via bug-fix agent, and refine successful code through refine agent; (3) Post-Train - construct prefix-completion pairs for post-training.

for LLM-generated unit tests. TestPilot (Schäfer et al., 2023) introduced a framework for automated test generation using LLMs. Several works focused on improving metrics like coverage and accuracy (Achiam et al., 2023; Ryan et al., 2024; Pizzorno and Berger, 2024).

Furthermore, some works considered integrating unit tests into LLM code generation to provide supervision for model-generated code during the inference phase. Works exemplified by Self-CodeAlign, Self-Edit, and Self-Debug (Wei et al., 2024a; Zhang et al., 2023; Chen et al., 2023) established test case-based interaction frameworks that enable LLMs to adjust generated code based on its accuracy against test cases.

Several works also leveraged unit tests as a code verification metric for synthesizing code data. AceCoder focuses on using unit tests as a filtering metric to synthesize preference dataset for training a reward model. KodCoder uses unit tests as a verification metric for post-training data synthesis. It first performs coding question synthesis on multiple subsets and then synthesizes data based on the generated instructions. rstar-coder, on the other hand, synthesizes a large number of code problems from a collection of high-quality, expert-written problems with oracle solutions, and then considers using auto-generated test cases as a verification metric.

In the UnitCoder framework, the Fix and Refine Flow stage is inspired by these approaches, implementing multiple rounds of refinement and code execution on raw code snippets that fail unit tests to improve the quality of the synthesized data. The core motivation behind UnitCoder is to directly leverage the inherent diversity of pre-training code corpora for post-training data synthesis, which represents an innovative attempt to bypass the need for synthesizing instruction data or relying on seed instruction data.

3 Method

In this section, we present UnitCoder, a scalable code synthesis pipeline that leverages pre-training code corpora and employs model-generated unit tests for both synthesis guidance and quality validation. The complete framework is illustrated in Figure 1.

The pipeline comprises three principal stages. In the first stage, we perform filtering of executable functions from a large-scale pre-trained code corpus. We then fine-tune a large language model to serve as our unit test generator, denoted as π_{θ_0}.

In the second stage, we build an iterative code

improvement framework with two key components: (i) a debugging agent that identifies and fixes potential defects in the original implementation through analysis of failed test cases and execution results, and (ii) a refinement agent that enhances code quality by adding docstrings and standardizing coding conventions once the code successfully passes the unit test.

In the post-training stage, we leverage the synthesized data to conduct post-training on open-source foundation models to validate the effectiveness of our approach.

3.1 Data Preparation

In the first stage of the UnitCoder pipeline, we filter executable function snippets from pre-training code corpus, and fine-tune a unit test generator to generate corresponding unit tests for the filtered functions.

3.1.1 Package-based Function Extraction

We extract executable code snippets from pre-training corpus through a two-step process: First, we perform AST-based semantic analysis to identify syntactically valid function units. Then, we filter these functions based on a predefined list of common APIs to retain those with meaningful package imports. This process yields a subset $\mathcal{D}_{pkg}$ from the original dataset $\mathcal{D}$.

3.1.2 Unit Test Generation

To generate corresponding unit tests for the extracted code snippets, we fine-tune a unit test generator that thoroughly evaluates complex function implementations. The generator creates comprehensive test cases to verify function behavior across edge cases, error conditions, and intricate API interactions. This generator, denoted as π_θ, is built upon Llama3-70B-Instruct and fine-tuned using high-quality function-test pairs. For each executable function $f_i \in \mathcal{D}_{pkg}$, π_{θ_0} generates a corresponding unit test u_i.

3.2 Fix and Refine Flow

In the second stage, we design an iterative code improvement framework based on unit test execution results. We utilize an open-source LLM to debug and fix code according to error traces, followed by a refinement step to ensure consistency in the quality of synthesized data. To ensure safe operation while processing code from unknown sources, we implement a security sandbox for code execution.

Algorithm 1 Code Improvement Pipeline

$\mathcal{D}_{pass} \leftarrow \emptyset$ ▷ Repository of validated code
$\mathcal{D}^0_{curr} \leftarrow \emptyset$ ▷ Queue of pending code
$r = 0$ ▷ Current iteration counter
$max\_round \in N$ ▷ Maximum iteration limit
Phase 1: Unit Test Initialization
for each function f_i^r in $\mathcal{D}_{p\_safe}$ **do**
  Generate comprehensive test suite for f_i^r
  $u_i \leftarrow \pi_{\theta_0}(f_i^r)$
  if f_i^r passes unit test u_i **then**
    Archive successfully validated code
    $\mathcal{D}_{pass} \leftarrow \mathcal{D}_{pass} \bigcup \{(f_i^r, u_i, r_i^r)\}$
  else
    Record execution diagnostics
    $\mathcal{D}^r_{curr} \leftarrow \mathcal{D}^r_{curr} \bigcup \{(f_i^r, u_i, r_i^r)\}$
  end if
end for
Phase 2: Iterative Code Improvement
$r = 1$
while $r \leq max\_round$ **do**
  $\mathcal{D}^r_{curr} \leftarrow \emptyset$
  for $(f_i^{r-1}, u_i, r_i^{r-1}) \in \mathcal{D}^{r-1}_{curr}$ **do**
    Apply improvement to f_i^{r-1}
    $f_i^r \leftarrow \pi_{\theta_1}(f_i^{r-1}, u_i, r_i^{r-1})$
    if f_i^r passes unit test u_i **then**
      $\mathcal{D}_{pass} \leftarrow D_{pass} \bigcup \{(f_i^r, u_i, r_i^r)\}$
    else
      $\mathcal{D}^r_{curr} \leftarrow D^r_{curr} \bigcup \{(f_i^r, u_i, r_i^r)\}$
    end if
    Collected functions that pass the unit tests.
  end for
  r ++
end while

3.2.1 Safety preparation

For safe execution, we build a secure sandbox, which redirects potentially risky operations, including file system operations like directory creation and deletion.

3.2.2 Iterative Code Improvement

After implementing security measures, we pair and execute functions along with their corresponding unit tests to obtain initial execution results. Specifically, for each function $f_i^0 \in \mathcal{D}_{pkg}$, its corresponding unit test is denoted as u_i. Functions that pass their unit tests are collected into a set $\mathcal{D}_{pass}$, while those that fail are placed in another set $\mathcal{D}^0_{curr}$ for subsequent iterative debugging.

Now that we collect the failed code snippets, the related unit tests, and the execution results, the bug-fix agent π_{θ_1} is employed to iteratively revise the failed codes. The complete process of code

improvement is detailed in Algorithm 1.

For the r-th iteration, let $\mathcal{D}_{curr}^{r-1}$ denote the collection of failed functions from round $r-1$. For each function $f_i^{r-1} \in \mathcal{D}_{curr}^{r-1}$ with its corresponding unit test u_i and execution result r_i^{r-1}, the revision step can be formulated as:

$$f_i^r = \pi_{\theta_1}(f_i^{r-1}, u_i, r_i^{r-1})$$

The revised function f_i^r is then evaluated using its corresponding unit test u_i. Functions that pass the unit tests are collected into $\mathcal{D}_{pass}$, while the failed ones are collected into $\mathcal{D}_{curr}^r$ for the next iteration. This iterative revision process continues until reaching the maximum iteration bound, accumulating all successfully fixed functions throughout the iterations.

3.2.3 Code Refinement

Through the iterative improvement process, we have constructed $\mathcal{D}_{pass}$, a collection of validated functions that pass their corresponding unit tests. Given that the original code corpus $\mathcal{D}$ is sourced from diverse repositories, it is necessary to normalize the coding style to ensure consistency in the synthetic data quality.

To address this requirement, we introduce a refine agent π_{θ_2} that enhances code readability in three aspects: (i) generating informative docstrings in natural language, (ii) adding explanatory inline comments at key code sections, and (iii) maintaining consistent coding style conventions.

3.3 Post-Train

In the third stage, we construct the post-training dataset $\mathcal{D}_{Unit}$ by reformulating the validated functions into supervised learning samples. Each training sample is structured as a pair, where the input consists of the import statements, function signature, and descriptive docstring, and the output contains the complete function implementation.

Subsequently, we conduct post-training on open-source foundation models using our synthetic dataset. Experimental results demonstrate both the diversity and high quality of our synthetic data, validating the effectiveness of the UnitCoder pipeline.

4 Experiment

In this section, we first briefly introduce the experimental setups, then discuss the experimental results, thoroughly demonstrating and validating the effectiveness of the UnitCoder pipeline.

4.1 Experimental Setups

Training Setup For the unit test generator π_{θ_0}, we employ Llama3-70B-Instruct as the foundation model. The post-training experiments are conducted on InternLM-2.5-7B and Llama-3.1-8B. We also perform ablation studies on the InternLM series to examine the impact of model scale. Both fine-tuning and post-training processes run for 1 epoch, with learning rates following a linear warmup and cosine decay schedule (1e-5 to 3e-6) and a maximum context window of 4096 tokens. The training utilizes A800 GPUs, with 64 GPUs for Llama3-70B-Instruct fine-tuning and 16 GPUs for smaller models.

Evaluation Setup We evaluate our post-trained models on three standard code benchmarks: BigCodeBench, HumanEval, and MBPP (Zhuo et al., 2024; Chen et al., 2021a; Austin et al., 2021), based on the OpenCompass framework (Contributors, 2023). The evaluation employs a 3-shot strategy for HumanEval and MBPP, while using complete mode for BigCodeBench. For unit test generator evaluation, we use solutions from HumanEval and MBPP as inputs to assess the accuracy of generated unit tests.

Unit Test Generator Setup We fine-tune the unit test generator π_{θ_0} based on Llama3-70B-Instruct. The fine-tuning data consists of unit test-function pairs from BigCodeBench, comprising 1140 functions with rich API calls and their corresponding human-written unit tests. To prevent evaluation set leakage, during supervised fine-tuning (SFT), we mask the original function when computing the loss.

Data Preparation In the UnitCoder pipeline, our pre-training code corpus primarily comes from The Stack pre-training dataset, where we have already performed data deduplication with evaluation benchmarks (e.g., HumanEval, MBPP, BigCodeBench, etc.). Additionally, we utilize an SFT dataset from WizardCoder (Luo et al., 2023), which serves as complementary data mixed with our synthetic data during the post-training stage, in order to maintains instruction-following capabilities.

To validate UnitCoder's effectiveness in complex API interactions, we compare against several synthetic datasets: **OSS-Instruct** (Wei et al., 2024c): A dataset of 75,000 instruction-code pairs synthesized from raw code. **OpenCoder-SFT-Stage-1** (Huang et al., 2024): A collection of 4.2M

	HumanEval	MBPP	BigCodeBench	BigCodeBench-Hard
Llama3.1-8B	36.6	58.8	31.0	5.4
+UnitCoder	**61.0**	**63.4**	**40.4**	**14.2**
InternLM2.5-7B	65.2	60.3	27.9	10.1
+UnitCoder	**67.1**	**66.2**	**39.3**	**17.6**
InternLM2.5-7B-Base	41.5	57.6	28.3	7.4
+UnitCoder	**62.2**	**65.8**	**41.6**	**14.9**

Table 1: Performance of base models post-trained with UnitCoder synthetic data. Results of BigcodeBench are tested under "complete" mode.

Models	BCB	BCB-Hard
Base Models (7B size)		
Llama3.1-8B	31.0	5.4
InternLM2.5-7B	27.9	10.1
InternLM2.5-7B-Base	28.3	7.4
Chat Models (7B size)		
Mistral-7B-Instruct-v0.3	25.7	6.8
Qwen2.5-7B-Instruct	42.4	14.2
Llam3.1-8B-Instruct	39.6	10.8
InternLM2.5-7B-Chat	32.9	5.4
Code LLMs (7B size)		
CodeLlama-7B-Instruct	27.3	4.1
Deepseek-Coder-6.7B	40.4	11.5
CodeQwen1.5-7B	43.4	14.8
Qwen2.5-Coder-7B	**45.3**	15.9
Ours (7B size)		
Llam3.1-8B+$\mathcal{D}_{Unit}$	40.4	14.2
InternLM2.5-7B-Base+$\mathcal{D}_{Unit}$	41.6	14.8
InternLM2.5-7B+$\mathcal{D}_{Unit}$	39.3	**17.6**

Table 2: Performance comparison between our proposed method and existing models on BigCodeBench (BCB) and BigCodeBench-Hard(BCB-Hard).

question-answer pairs spanning diverse computer science domains, generated from general code corpora. **Evol-codealpaca-v1** (Luo et al., 2023): A dataset of 110K instruction pairs created by augmenting instructions using GPT-4.

4.2 Post-Training Performance Analysis

Table 1 demonstrates the effectiveness of UnitCoder in enhancing LLMs' code capabilities. Our post-training approach combines synthetic data and SFT data, and achieves significant improvements across all base models.

On the BigCodeBench benchmark, which evaluates package calling capabilities in complex scenarios, UnitCoder significantly improves the performance of multiple base models: InternLM 2.5-7B's accuracy increases from 27.9% to 39.3%, InternLM 2.5-7B-base from 28.3% to 41.6%, and Llama3.1-8B from 31.0% to 40.4%. Furthermore, post-trained base models demonstrate consistent improvements across other code benchmarks, including HumanEval and MBPP. These comprehensive performance gains across multiple benchmarks validate the effectiveness of the UnitCoder approach.

Method	BCB	BCB-Hard
Llama3.1-8B	31.0	5.4
+Evol	26.2	6.1
+OSS	27.5	6.8
+OpenCoder	32.3	10.1
+UnitCoder	**40.4**	**14.2**
InternLM2.5-7B	27.9	10.1
+Evol	21.1	4.1
+OSS	22.8	5.4
+OpenCoder	28.2	8.8
+UnitCoder	**39.3**	**17.6**

Table 3: Performance comparison between UnitCoder dataset and other synthetic datasets on BigCodeBench (BCB) and BigCodeBench-Hard(BCB-Hard). Evol, OpenCoder and OSS-Instruct refer to Evol-codealpaca-v1, OpenCoder-SFT-Stage-1 and OSS-Instruct-75K datasets, respectively.

4.3 Analysis of Comparative Experiments

We conduct extensive comparative experiments on BigCodeBench to comprehensively evaluate the effectiveness of our approach on complex API invocation tasks. Table 2 presents comparisons among 7B-scale models, including base models, instruction-tuned models, and code-specialized models. Our method achieves comparable performance to leading instruction-tuned models, and significantly outperforms mainstream pre-trained models. Notably, on BigCodeBench-Hard, which evaluates complex API composition capabilities, our approach matches or even exceeds the performance of code-

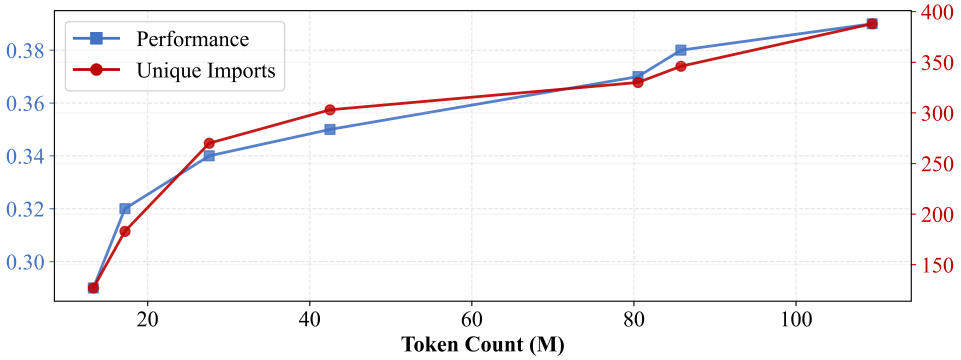

Figure 2: Scaling Effects of Synthetic Data: As the scale of synthetic data (measured in tokens) increases, we observe a corresponding growth in both the diversity of unique packages in synthetic data and InternLM2.5-7B's performance on BigCodeBench after post-training.

Method	HumanEval	MBPP	BigCodeBench	BigCodeBench-Hard
Base Model	65.2	60.3	27.9	10.1
+ General Code	58.5	61.1	29.4	7.4
+ $\mathcal{D}_{pkg}$	50.6	54.5	29.7	4.1
+ General Code + $\mathcal{D}_{pkg}$	61.0	62.3	31.1	6.1
+ General Code + $\mathcal{D}_{pass}$	61.6	61.1	35.2	13.5
+ General Code + $\mathcal{D}_{Unit}$**(Ours)**	**67.1**	**66.2**	**39.3**	**17.6**

Table 4: Ablation study of the UnitCoder pipeline, showing performance comparison of InternLM-2.5-7B under different training configurations. The evaluation demonstrates the impact of various training data combinations: general code data (General SFT dataset), $\mathcal{D}_{pkg}$(package-centric subset without verification), $\mathcal{D}_{pass}$(Verified dataset without refine), and $\mathcal{D}_{Unit}$(verified and refined data generated through the UnitCoder pipeline).

specialized models of similar size.

To further evaluate the effectiveness of our approach, we conducted controlled experiments by fine-tuning the same base model with different training datasets, as shown in Table 3. The results demonstrate that our method achieves the most significant performance improvements on BigCodeBench among all compared approaches. This superior performance on API-related tasks clearly validates the quality of our synthetic dataset and the effectiveness of the UnitCoder framework, especially considering these improvements were achieved with a relatively compact dataset.

4.4 Scaling Effects of Synthetic Data

For investigating the impact of synthetic data scale on model performance, we conduct a series of controlled experiments with increasing data size. We first analyze the occurrence distribution of different APIs in the original code corpus, as shown in Figure 3. The results demonstrate a distinct long-tail distribution, where a small number of frequently used packages account for the majority of invocations, while most packages exhibit relatively low occurrence frequencies.

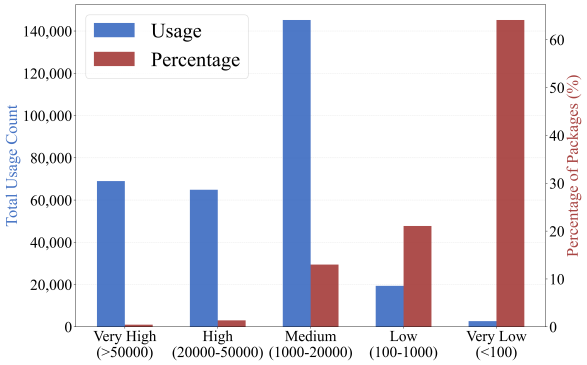

Figure 3: The distribution of packages in filtered code data, grouped by usage frequency. Usage represents the frequency of package imports, and Percentage shows the percentage of package types within each frequency group relative to the total number.

For packages with lower occurrence frequencies, UnitCoder demonstrates effective verification and synthesis of high-quality data. As shown in Figure 2, the expansion of synthetic data scale leads to two significant improvements: First, it enables the capture of a broader spectrum of API call patterns, particularly those that appear infrequently in the original corpus. Second, it contributes to consis-

	BigCodeBench	BigCodeBench-Hard
InternLM2.5-1.8B	14.7	2.0
+UnitCoder	**19.6**	**4.1**
InternLM2.5-7B	27.9	10.1
+UnitCoder	**39.3**	**17.6**
InternLM2.5-20B	41.1	14.2
+UnitCoder	**44.6**	**22.3**

Table 5: Abaltion study on model scale.

Benchmark	**Accuracy**	**Coverage**
HumanEval	80.4	96.9
MBPP	84.2	92.5
FullStackBench	65.2	82.6
DS-1000	66.7	98.7
LiveCodeBench	73.5	87.9

Table 6: Unit test generator evaluation

tent performance improvements on BigCodebench, where complex package usage is needed.

4.5 Ablation Studies

In this section, we present ablation studies to comprehensively evaluate the key components of our UnitCoder pipeline. Our experiments address three critical research questions: (i) the necessity of the iterative verification process, (ii) the impact of the refine agent, and (iii) the consistency of synthetic data's effectiveness across different model scales. Our experimental results are presented in Table 4 and Table 5.

RQ1: How essential is the verification pipeline? We first evaluate the effectiveness of the iterative code improvement module. As reported in Table 4, we first compare two experimental settings: (i) fine-tuning with general SFT data alone, and (ii) fine-tuning with a combination of SFT data and unit-test verified data ($\mathcal{D}_{pass}$).

Results show that the verification process brings substantial performance gains across all benchmarks, with the most notable improvement observed on BigCodeBench where the pass rate increases from 29.4% to 35.2%. This demonstrates the critical role of verification in enhancing model performance, even before applying subsequent refinement steps.

To further examine whether similar performance gains could be achieved without verification, we conduct a comparative experiment using unverified package-centric data ($\mathcal{D}_{pkg}$) combined with SFT data. The results show that our verification process yields a 4% performance improvement on BigCodeBench and a substantial 7% gain on BigCodeBench-Hard. These performance gains clearly demonstrate that the verification process is an irreplacable component of our pipeline.

RQ2: Is code refinement necessary? Following our verification pipeline analysis, we investigate the effectiveness of the refinement process through ablation studies in Table 4. We compare two experimental settings: (i) fine-tuning with a combination of general SFT data and UnitCoder-synthesized data, and (ii) fine-tuning with SFT data combined with verified-only data ($\mathcal{D}_{pass}$). Our results show consistent performance improvements across all benchmarks after applying the refinement process.

Notably, compared to the $\mathcal{D}_{pass}$ mixture method, we observe particularly significant improvements on HumanEval and MBPP benchmarks, which focus less on complex package interactions. These results indicate that the refinement step comprehensively enhances the quality of synthetic data, enabling the model to better learn fundamental coding capabilities from the original codebase.

RQ3: Does the method work on different model scales? To further investigate the impact of our synthetic data, we trained models from the InternLM2.5 series across different scales. Table 5 reports our results, demonstrating consistent performance improvements across all model sizes. Specifically, on BigCodeBench, the 1.8B variant improves from 14.6% to 19.6%, the 7B variant from 27.9% to 39.3%, and the 20B variant from 41.1% to 44.6%. Furthermore, on BigCodeBench-Hard, our synthetic data brings an average improvement of approximately 6% across all model scales. These results highlight how effectively the synthetic data enhances performance on package-related coding tasks across various model sizes.

4.6 Unit Test Generator Evaluation

To assess our unit test generator, we conduct experiments on benchmarks including HumanEval (Chen et al., 2021a), MBPP (Austin et al., 2021), FullStackBench (Cheng et al., 2024), DS-1000 (Lai et al., 2023), and LiveCodeBench (Jain et al., 2024).

We use canonical solutions as input functions and evaluate whether the generated unit tests effectively validate these functions. As shown in Table 6, our generator performs well at validating

code in scenarios with rich function calls.

However, compared to simpler, competition-level code datasets like HumanEval and MBPP, the generator shows a decrease in both line coverage and accuracy on datasets with more complex function-calling scenarios. This observation confirms that the current unit test generator has room for improvement when dealing with more complex datasets, real-world applications, and unseen function-calling situations.

Even with the current unit test generator that still has room for improvement, the UnitCoder framework has demonstrated high synthesis quality, which we believe proves the effectiveness of the UnitCoder framework and our posposed methodology.

5 Conclusion

We present UnitCoder, a scalable framework for synthesizing high-quality post-training code data from raw code corpora under unit test guidance. The framework innovatively leverages code executability through unit tests as the primary guidance, ensuring the synthesis of high-quality data while preserving the original code functionality. By synthesizing a dataset of over 500K verifiable programs, we demonstrate through extensive experiments that our synthetic data consistently improves models' performance on code generation benchmarks, particularly in handling complex API interactions. Through comprehensive ablation studies, we validate each component's necessity and analyze the relationships between data scale, diversity, and model performance, providing valuable insights for scalable code synthesis. We believe UnitCoder demonstrates an effective approach for scalable, high-quality code data synthesis, providing valuable insights for future research in LLM-based code generation.

Limitations

Despite the demonstrated effectiveness of UnitCoder, our approach has several limitations that warrant discussion. First, while UnitCoder shows promising results with our current unit test generator, utilizing more advanced models could potentially improve synthesis quality. The trade-off between model capabilities and computational efficiency requires further investigation. Furthermore, our framework is currently limited to Python code synthesis. Extending UnitCoder to multiple programming languages would help validate its generalizability across different development contexts.

References

OpenAI Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mo Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Benjamin Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Sim'on Posada Fishman, Juston Forte, Is abella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Raphael Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Lukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Hendrik Kirchner, Jamie Ryan Kiros, Matthew Knight, Daniel Kokotajlo, Lukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Ma teusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel P. Mossing, Tong Mu, Mira Murati, Oleg Murk, David M'ely, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Ouyang Long, Cullen O'Keefe, Jakub W. Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alexandre Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres,

Michael Petrov, Henrique Pondé de Oliveira Pinto, Michael Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack W. Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario D. Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin D. Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas A. Tezak, Madeleine Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cer'on Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll L. Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qim ing Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. 2023. Gpt-4 technical report.

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*.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenhang Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, K. Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Yu Bowen, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xing Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. 2023. Qwen technical report. *ArXiv*, abs/2309.16609.

Zheng Cai, Maosong Cao, Haojiong Chen, Kai Chen, Keyu Chen, Xin Chen, Xun Chen, Zehui Chen, Zhi Chen, Pei Chu, et al. 2024. Internlm2 technical report. *arXiv preprint arXiv:2403.17297*.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*.

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

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé, Jared Kaplan, Harrison Edwards, Yura 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, David W. Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William H. Guss, Alex Nichol, Igor Babuschkin, Suchir Balaji, Shantanu Jain, Andrew Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew M. Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021b. Evaluating large language models trained on code. *ArXiv*, abs/2107.03374.

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

Yao Cheng, Jianfeng Chen, Jie Chen, Li Chen, Liyu Chen, Wentao Chen, Zhengyu Chen, Shijie Geng, Aoyan Li, Bo Li, et al. 2024. Fullstack bench: Evaluating llms as full stack coders. *arXiv preprint arXiv:2412.00535*.

OpenCompass Contributors. 2023. Opencompass: A universal evaluation platform for foundation models. https://github.com/open-compass/opencompass.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing

Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Preprint*, arXiv:2501.12948.

DeepSeek-AI, Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Hao Yang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jin Chen, Jingyang Yuan, Junjie Qiu, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruizhe Pan, Runxin Xu, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Size Zheng, T. Wang, Tian Pei, Tian Yuan, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Liu, Xin Xie, Xingkai Yu, Xinnan Song, Xinyi Zhou, Xinyu Yang, Xuan Lu, Xuecheng Su, Y. Wu, Y. K. Li, Y. X. Wei, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Zheng, Yichao Zhang, Yiliang Xiong, Yilong Zhao, Ying He, Ying Tang, Yishi Piao, Yixin Dong, Yixuan Tan, Yiyuan Liu, Yongji Wang, Yongqiang Guo, Yuchen Zhu, Yuduan Wang, Yuheng Zou, Yukun Zha, Yunxian Ma, Yuting Yan, Yuxiang You, Yuxuan Liu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhewen Hao, Zhihong Shao, Zhiniu Wen, Zhipeng Xu, Zhongyu Zhang, Zhuoshu Li, Zihan Wang, Zihui Gu, Zilin Li, and Ziwei Xie. 2024. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *Preprint*, arXiv:2405.04434.

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

Huawen Feng, Pu Zhao, Qingfeng Sun, Can Xu, Fangkai Yang, Lu Wang, Qianli Ma, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, et al. 2024. Warriorcoder: Learning from expert battles to augment code large language models. *arXiv preprint arXiv:2412.17395*.

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*.

Dong Huang, Qingwen Bu, Jie M. Zhang, Michael Luck, and Heming Cui. 2023. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *ArXiv*, abs/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. 2024. 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, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-coder technical report. *ArXiv*, abs/2409.12186.

Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. Mapcoder: Multi-agent code generation for competitive problem solving. *arXiv preprint arXiv:2405.11403*.

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

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

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. Ds-1000: A

natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

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

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. *ICLR*.

Juan Altmayer Pizzorno and Emery D Berger. 2024. Coverup: Coverage-guided llm-based test generation. *arXiv preprint arXiv:2403.16218*.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, I. Evtimov, Joanna Bitton, Manish P Bhatt, Cristian Cantón Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre D'efossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code. *ArXiv*, abs/2308.12950.

Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024. Code-aware prompting: A study of coverage-guided test generation in regression setting using llm. *Proceedings of the ACM on Software Engineering*, 1(FSE):951–971.

Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*.

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. *arXiv preprint arXiv:2410.24198*.

Yuxiang Wei, Hojae Han, and Rajhans Samdani. 2024b. Arctic-snowcoder: Demystifying high-quality data in code pretraining. *Preprint*, arXiv:2409.02326.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024c. Magicoder: Empowering code generation with oss-instruct. *Preprint*, arXiv:2312.02120.

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Ke-Yang Chen, Kexin Yang, Mei Li, Min Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Yang Fan, Yang Yao, Yichang Zhang, Yunyang Wan, Yunfei Chu, Zeyu Cui, Zhenru Zhang, and Zhi-Wei Fan. 2024a. Qwen2 technical report. *ArXiv*, abs/2407.10671.

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, et al. 2024b. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*.

Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiufeng Yin. 2023. Wavecoder: Widespread and versatile enhancement for code large language models by instruction tuning. In *Annual Meeting of the Association for Computational Linguistics*.

Daoguang Zan, Ailun Yu, Bo Shen, Jiaxin Zhang, Taihong Chen, Bing Geng, B. Chen, Jichuan Ji, Yafen Yao, Yongji Wang, and Qianxiang Wang. 2023. Can programming languages boost each other via instruction tuning? *ArXiv*, abs/2308.16824.

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. *arXiv preprint arXiv:2410.05605*.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-edit: Fault-aware code editor for code generation. *arXiv preprint arXiv:2305.04087*.

Qingfu Zhu, Xianzhen Luo, Fang Liu, Cuiyun Gao, and Wanxiang Che. 2022. A survey on natural language processing for programming. *ArXiv*, abs/2212.05773.

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

A More Discussions

A.1 Experiments on Code Specialized Models

Although UnitCoder is designed for generating post-training data for base models, to provide a more comprehensive evaluation of model performance, we supplement our experiments with

Model	BigCodeBench-Hard
Qwen-2.5-Coder-7B-Base	15.9
+UnitCoder (1 epoch)	**16.9**
DeepSeek-Coder-6.7B-Base	11.5
+UnitCoder (1 epoch)	**15.5**

Table 7: Performance improvement with UnitCoder on code-specialized base models.

Dataset	Line Coverage
Subset-2K	**97.2**

Table 8: Line coverage of the unit test generator on raw code corpus.

results from further fine-tuning code-specialized models, as shown in Table 7. We observe that our synthetic data can further improve the performance of code-optimized models like Qwen2.5-7B-Coder-Base and DeepSeek-Coder-6.7B-Base on BigCodeBench-Hard, indicating that our data can still enhance the capabilities of already powerful code-specialized models on complex API calling tasks.

A.2 Experiments on Unit Test Generator

We further test the coverage of our unit test generator on a subset of the original code corpus. Specifically, we randomly sample 2000 code snippets from the original code dataset after filtering, and test the line coverage of our unit test generator on these code snippets. The results are shown in Table 8.

A.3 Further Statistics of the Post-training Dataset

To complement our analysis of function call distribution, we provide additional statistics on the post-training dataset, as shown in Table 8. Overall, the statistics reveal several key characteristics. The average length of the function completions is approximately 17 lines, while the average prompt length is around 24 lines. On average, each data sample includes at least one import statement, and the total number of unique function call types reaches 373.

A.4 Performance of Code Specialized LLMs

To provide a more comprehensive performance comparison, we also evaluate the performance of several code-specialized LLMs on HumanEval and MBPP in Table 10.

Table 9: Statistics of the Post-training Dataset

Metric	Value
Total Number of Samples	500,000
Average Prompt Lines	24.2
Average Output Lines	17.1
Average Prompt Tokens	196.9
Average Output Tokens	164.7
Average Import Count	1.1
Unique Import Count	373

Table 10: Performance of Code-Specialized LLMs on HumanEval and MBPP (pass@1)

Model	HumanEval	MBPP
Qwen2.5-Coder	88.4	83.5
DeepSeek-Coder	78.6	65.4

B Demonstration of the Pipeline

In this section, we demonstrate our synthetic pipeline through several examples. The demonstrations include:

- A demonstration of unit test generation, showing the original function, its generated unit tests, and the test execution output format.
- The bug-fixing process, demonstrating how the system handles failed test cases.
- The complete refined function after all iterations and improvements.
- Prompts for the unit test generator and the agents.

Unit Test Demo

Function:

```
import numpy as np

def drawWeights(size, distribution):
    # Validate the size parameter
     if not isinstance(size, (int, tuple)) or (isinstance(size, int) and size <=
0):
         raise ValueError("Size must be a positive integer or a tuple of positive
integers.")

    # Validate the distribution parameter
    if distribution not in ['lognormal', 'normal', 'uniform']:
         raise ValueError("Distribution must be one of 'lognormal', 'normal', or
'uniform'.")

    weights = None
    if distribution == 'lognormal':
        hyp = 1.0
        m = np.log(0.2) + hyp
        s = hyp
        weights = (np.random.lognormal(m, s, size) * (255 / 20.0)).astype(int)
    elif distribution == 'normal':
        m = 10
        s = 5
        weights = np.random.normal(m, s, size).astype(int)
    elif distribution == 'uniform':
        weights = np.random.uniform(0, 255, size).astype(int)

    return weights
```

Unit Test:

```
import unittest
import numpy as np
class TestCases(unittest.TestCase):
    def test_lognormal_weights(self):
        np.random.seed(42)
        weights = drawWeights(10, 'lognormal')
        self.assertIsInstance(weights, np.ndarray)
        self.assertEqual(len(weights), 10)
        self.assertTrue(all(0 <= w <= 255 for w in weights))
    def test_normal_weights(self):
        np.random.seed(42)
        weights = drawWeights(10, 'normal')
        self.assertIsInstance(weights, np.ndarray)
        self.assertEqual(len(weights), 10)
        self.assertTrue(all(-5 <= w <= 20 for w in weights))
    def test_uniform_weights(self):
        np.random.seed(42)
        weights = drawWeights(10, 'uniform')
        self.assertIsInstance(weights, np.ndarray)
        self.assertEqual(len(weights), 10)
        self.assertTrue(all(0 <= w <= 255 for w in weights))
    def test_invalid_size(self):
        with self.assertRaises(ValueError):
            drawWeights(-1, 'lognormal')
    def test_invalid_distribution(self):
        with self.assertRaises(ValueError):
            drawWeights(10, 'invalid_distribution')
```

Execution Result:

```
["pass",{}]
```

Table 11: Demo of a function, its corresponding unit test, and the execution result.

Debug Demo

```
Failed Test Case:
def test_data_file_with_non_image_entries(self):
    # Create a data file with non-image entries
    mixed_data_file_path = os.path.join(self.test_dir, "mixed_data.txt")
    with open(mixed_data_file_path, 'wt') as f:
        f.write("input_image3.png annotation_image3.png\n")
        f.write("non_image_data.txt\n")
    # Try to load the mixed data file
    with self.assertRaises(ValueError):
        _load_data_files(mixed_data_file_path)
```

```
Original Execution Result:
[
    "fail",
    {
                    "test_data_file_with_non_image_entries":   "Traceback   (most
recent    call    last):\n        File    \"__test__.py\",    line    140,    in
test_data_file_with_non_image_entries\nAssertionError: ValueError not raised\n",
        "test_invalid_data_file_format": "Traceback (most recent call last):\n
File \"__test__.py\", line 117, in test_invalid_data_file_format\nAssertionError:
ValueError not raised\n"
    }
]
```

```
Fix Bug Accorrdingly:
import numpy as np
import os
import logging
# Configure logging
logging.basicConfig(level=logging.INFO,  format='%(asctime)s  -  %(levelname)s  -
%(message)s')
def _load_data_files(data_file_path):
    input_path_list = []
    annotation_path_list = []
    # Check if the data file exists
    if not os.path.exists(data_file_path):
        logging.error(f"Data file not found: {data_file_path}")
        raise FileNotFoundError(f"Data file not found: {data_file_path}")
    data_folder_path = os.path.dirname(os.path.abspath(data_file_path))
    try:
        with open(data_file_path, 'rt') as f:
            for line in f:
                # Split the line into input and annotation paths
                parts = line.strip().split(' ')
                if len(parts) != 2:
                    logging.warning(f"Skipping invalid line: {line.strip()}")
                    raise ValueError(f"Invalid line format: {line.strip()}")
                input_path, annotation_path = parts
                # Construct full paths
                full_input_path = os.path.join(data_folder_path, input_path)
            full_annotation_path = os.path.join(data_folder_path, annotation_path)
                # Validate paths
                            if  not  os.path.exists(full_input_path)  or  not
os.path.exists(full_annotation_path):
                logging.warning(f"Skipping non-existent paths: {full_input_path}
or {full_annotation_path}")
                    raise ValueError(f"Non-existent paths: {full_input_path} or
{full_annotation_path}")  ##### ValueError raise added here #####
                input_path_list.append(full_input_path)
                annotation_path_list.append(full_annotation_path)
    except Exception as e:
        logging.error(f"Error reading data file: {e}")
        raise
    return np.array(input_path_list), np.array(annotation_path_list)
```

Table 12: Demo of the code enhancement pipeline.

Refine Demo

```
import numpy as np

def get_var(data):
    """
    Calculates the variance of a given list of numbers.

    Parameters:
   - data (list of float or int): A list of numerical values for which to calculate
the variance.

    Returns:
    - float: The variance of the input data.

    Requirements:
    - numpy

    Example:
    >>> var = get_var([1, 2, 3, 4, 5])
    >>> print(var)
    2.0
    """
    # Calculate the mean of the data
    mean = np.mean(data)

    # Calculate the variance using the formula: sum((x - mean)^2) / n
    var = sum([np.power(x - mean, 2) for x in data]) / len(data)

    return var
```

Table 13: Demo of the code snippet after refinement.

Prompt for the unit test generator

````
You are a professional software testing expert. Your task is to write comprehensive
unit tests for the given function.

Please follow these guidelines:
1. Write tests that cover different scenarios including:
   - Normal/expected inputs
   - Edge cases
   - Invalid/unexpected inputs
   - Boundary conditions

2. Each test case should:
   - Have a clear and descriptive name
   - Include assertions that verify both return values and expected behavior
   - Be independent of other test cases
   - Include brief comments explaining the test purpose

3. Test structure requirements:
   - Use the unittest framework
   - Create a proper test class inheriting from unittest.TestCase
   - Include setUp/tearDown methods if necessary
   - Write self-contained tests that don't rely on external resources

4. Important:
   - Only output the test code within Python code blocks
   - Ensure all necessary imports are included
   - Focus on functionality testing rather than implementation details
   - Write tests that are maintainable and readable

Please analyze the given function and generate appropriate unit tests following
these guidelines.
Your output format should be like this:
```python
import unittest
class TestCases(unittest.TestCase):
    def test_case_1(self):
        # Test purpose: Verify the function handles normal inputs correctly
        self.assertEqual(function_name(input1, input2), expected_output1)
    def test_case_2(self):
        ...
```
Do not modify the class name(TestCases).
````

Table 14: Prompt for unit test generator.

Prompt for the bug-fix agent.

````
You are a powerful coding expert specialized in code debugging and optimization.
Your task is to fix the given code based on unit test results and error messages.

Please follow these guidelines:
1. Carefully analyze:
   - The original code implementation
   - Failed test cases and their error messages
   - Test requirements and expected behavior

2. When fixing the code:
   - Make minimal necessary changes to fix the issues
   - Maintain the original code structure when possible
   - Ensure the solution is efficient and clean

3. Important:
   - Only output the fixed code within Python code blocks
   - Ensure the solution passes all test cases
   - Focus on addressing the specific test failures
   - Maintain code readability and best practices

Please analyze the code and test failures, then provide the corrected
implementation.
Your output format should be like this:
```python
# imports
def function_name(params):
    # Fixed implementation
    ...
```
````

Table 15: Prompt for bug-fix agent.

Prompt for the refine agent.

````
You are a powerful coding expert specialized in code documentation and optimization.
Given a code snippet and its unit tests, please enhance the code with comprehensive
documentation while maintaining its functionality.

Requirements:
1. Documentation Enhancement:
   - Add clear function description
   - Document parameters and return values
   - List required dependencies
   - Provide usage examples
   - Document potential exceptions (if applicable)

2. Code Refinement Guidelines:
   - Add concise inline comments at key points
   - Maintain code functionality
   - Ensure code remains readable and well-styled
   - Add necessary error handling without affecting core logic
   - Keep function names unchanged

3. Documentation Format:
   - Function description
   - Parameters
   - Returns
   - Requirements
   - Raises (if applicable)
   - Examples

Your output should follow this structure:
```python
def function_name(params):
    # Core documentation
    # Implementation with inline comments
    ...
```
````

Table 16: Prompt for refine agent.