KodCode : A Diverse, Challenging, and Verifiable Synthetic Dataset for Coding

Zhangchen Xu[♣*] **Yang Liu**[◇] **Yueqin Yin**[♠*]
Mingyuan Zhou[♠] **Radha Poovendran**[♣]

[◇]Microsoft GenAI [♣]University of Washington [♠]The University of Texas at Austin

{zxu9,rp3}@uw.edu, yaliu10@microsoft.com, {yueqin.yin, mingyuan.zhou}@utexas.edu

https://kodcode-ai.github.io 🤗 https://huggingface.co/KodCode

Abstract

We introduce KodCode, a synthetic dataset that addresses the persistent challenge of acquiring high-quality, verifiable training data across diverse difficulties and domains for training Large Language Models for coding. Existing code-focused resources typically fail to ensure either the breadth of coverage (e.g., spanning simple coding tasks to advanced algorithmic problems) or verifiable correctness (e.g., unit tests). In contrast, KodCode comprises question–solution–test triplets that are systematically validated via a self-verification procedure. Our pipeline begins by synthesizing a broad range of coding questions, then generates solutions and test cases with additional attempts allocated to challenging problems. Finally, post-training data synthesis is done by rewriting questions into diverse formats and generating responses under a test-based reject sampling procedure from a reasoning model (DeepSeek R1). This pipeline yields a large-scale, robust and diverse coding dataset. KodCode is suitable for supervised fine-tuning and the paired unit tests also provide great potential for RL tuning. Fine-tuning experiments on coding benchmarks (HumanEval(+), MBPP(+), BigCodeBench, and LiveCodeBench) demonstrate that KodCode-tuned models achieve state-of-the-art performance, surpassing models like Qwen2.5-Coder-32B-Instruct and DeepSeek-R1-Distill-Llama-70B.

1 Introduction

Recent advances in Large Language Models (LLMs) for coding such as Qwen2.5-Coder (Hui et al., 2024), Deepseek Coder (Guo et al., 2024), and OpenCoder (Huang et al., 2024) have demonstrated remarkable capabilities in programming tasks. These models excel at function writing (Chen et al., 2021), debugging (Zhong et al., 2024), issue resolution (Zhang et al., 2024b), and agent system enhancement (Zhang et al., 2024a), fundamentally transforming software development practices (Qian et al., 2024; Hou et al., 2024).

Ideally, training high-performing coding LLMs requires high-quality data with verified solutions and test cases for post-training stages, including supervised fine-tuning (SFT) and reinforcement learning (RL) (DeepSeek-AI et al., 2025; Team, 2025; Hui et al., 2024; Wei et al., 2024a). While human-curated coding datasets like TACO (Li et al., 2023), APPS (Hendrycks et al., 2021), and CodeContests (Li et al., 2022) offer high quality questions, canonical solutions, and tests, their limited scale constrains model training. Synthetic datasets have emerged as an alternative (Wang et al., 2023; Long et al., 2024), but often lack diversity (Xu et al., 2024), sufficient complexity (Luo et al., 2023), and reliable response verification (Lei et al., 2024).

In this paper, we bridge this gap by introducing KodCode-V1, hereafter referred to as KodCode, a synthetic dataset consisting of 447K coding questions with verified solutions and unit tests. Our approach starts from synthesizing coding questions from 12 sources using five distinct methods to ensure diversity and complexity. In solution & test generation step, we generate unit tests along with

Dataset Name	#Problems	Diversity	Difficulty	Unit Test	Verified Solution
APPS (Hendrycks et al., 2021)	10K	High	High	●	●
CodeContests (Li et al., 2022)	13K	High	High	●	●
TACO (Li et al., 2023)	26K	High	High	●	●
Code Alpaca (Chaudhary, 2023)	20K	Low	Low	○	○
SelfCodeAlign (Wei et al., 2024a)	50K	Mid	Low	○	●
OSS Instruct (Wei et al., 2024c)	75K	Mid	Mid	○	○
AceCoder (Zeng et al., 2025)	87K	Mid	Mid	●	○
Evol Instruct (Luo et al., 2023)	111K	Low	Mid	○	○
Educational Instruct (Huang et al., 2024)	118K	Low	Low	●	●
Package Instruct (Huang et al., 2024)	171K	Mid	Mid	○	○
KodCode -V1	**447K**	High	Mix	●	●

Table 1: Comparison of KodCode with existing code datasets for LLM post-training. The first three rows show human-curated datasets, while the remaining rows represent synthetic datasets. KodCode offers three difficulty labels (e.g., "easy", "medium", and "hard"), which we denote as "Mix".

[*]Work done during internship at Microsoft GenAI.

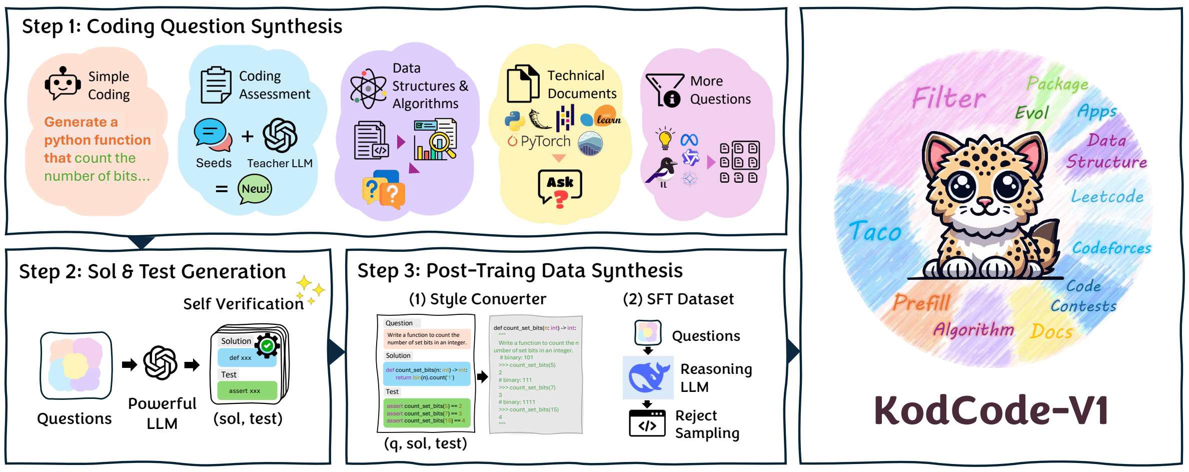

Figure 1: This figure demonstrates the pipeline for generating KodCode-V1. Our approach follows a three-step pipeline: *Coding Question Synthesis*, *Solution & Test Generation*, and *Post-training Data Synthesis*. The final KodCode-V1 dataset contains 447K verified question-solution-test triplets. The distribution of each subset is demonstrated on the right.

the solution and execute the unit test to verify the correctness of the solution. This *self-verification* mechanism not only ensures each solution is functionally correct but also offers verifiable correctness by providing explicitly curated unit tests. In addition, for challenging questions whose solutions fail the self-verification, we allocate additional attempts rather than discarding them, ensuring challenging questions are not filtered out during reject sampling. We further enhance the dataset for post-training by rewriting questions into diverse formats and generating chain-of-thought (CoT) responses using DeepSeek-R1 (DeepSeek-AI et al., 2025) under a test-based reject sampling procedure.

We conduct comprehensive analyses of our coding data generation pipeline. First, we evaluate our self-verification mechanism by testing solutions against human-written unit tests from the MBPP validation dataset. Our experiments show that error rate remains below 2.5%, demonstrating the effectiveness of our self-verification mechanism. We then examine the benefits of scaled computation for challenging coding questions. Finally, we perform statistical analyses of KodCode's token length, diversity, difficulty distribution, and potential contamination with existing benchmarks.

Furthermore, to validate KodCode's effectiveness for code LLM post-training, we evaluate models fine-tuned on KodCode across standard benchmarks: HumanEval(+), MBPP(+), BigCodeBench, and LiveCodeBench. Our experimental results show that KodCode-fine-tuned models achieve state-of-the-art performance, surpassing other open-source models such as Qwen2.5-Coder-32B-Instruct and DeepSeek-R1-Distill-Llama-70B in most of the benchmarks.

We hope our open-source dataset and models will help the community develop more capable coding assistants. We believe KodCode will advance current SFT and RL post-training pipelines for code generation models, pushing the boundaries of LLMs in coding tasks.

2 KodCode : Synthesizing Diverse, Challenging, and Verifiable Correct Post-Training Data for Code

As illustrated in Figure 1, our approach follows a three-step pipeline: *Coding Question Synthesis*, *Solution & Test Generation*, and *Post-training Data Synthesis*. Generally, we begin by synthesizing diverse coding questions q through a combination of prompt engineering and LLM-based augmentation. Next, we leverage a self-verification process to create high-quality solutions and test cases $(sol, test)$ while offering verifiable correctness. Finally, to provide high-quality post-training data, we diversify the generated synthetic questions by rewriting them into different formats, and generate responses by prompting a reasoning model (i.e., DeepSeek-R1) with a test-based reject sampling procedure.

2.1 Step 1: Coding Question Synthesis

To generate challenging coding questions with broad coverage, we developed 12 distinct subsets spanning various domains (from algorithmic to package-specific knowledge) and difficulty levels (from basic coding exercises to interview and competitive programming challenges). Below, we elaborate on the pipeline used to construct each subset.

Simple Coding Questions. To generate simple coding questions, we extend the MAGPIE framework (Xu et al., 2024) and introduce **MAGPIE-Prefill**. This approach generates simple coding questions by prefilling the user message in the chat template with a pre-defined suffix (e.g., *"Write a Python function that"*), and leverages *Qwen2.5-Coder-7B-Instruct* to complete the remaining part of the user query. This method efficiently generates diverse questions focused on function implementation and basic Python programming tasks. We name this subset **Prefill**. The complete prompt template is provided in Appendix D.2.

Coding Assessment Questions. To synthesize diverse coding assessment questions, we leverage existing human-written coding assessment datasets as seed corpora. To expand these datasets, we employ *GPT-4o-0513* as a teacher LLM, prompting it to act as an expert programming instructor. The model analyzes the structure, complexity, and knowledge requirements of seed questions and subsequently generates new assessment questions that maintain consistency in difficulty and scope.

The seed datasets utilized include LeetCode (Hartford, 2023), Codeforces (Jur1cek, 2022), APPS training subset (Hendrycks et al., 2021), TACO training subset (Li et al., 2023), and Code Contests (Li et al., 2022). The respective question subsets are named **LeetCode**, **Codeforces**, **APPS**, **Taco**, and **Code Contests**. The complete prompt templates are shown in Appendix D.3.

Data Structures and Algorithms. While coding assessments typically focus on specific programming concepts, they do not fully encompass Data Structures and Algorithms (DSA) knowledge. To bridge this gap, we convert Python DSA knowledge into assessment questions by uniformly sampling a collection of DSA code snippets (The Algorithms, 2023; Keon, 2018). We prompt LLM to first perform a systematic analysis of DSA snippets (addressing core components, complexity, and implementation challenges), then craft questions that test foundational understanding to avoid direct code replication. We denote the two subsets generated using this method as **Algorithm** and **Data Structure**. The corresponding prompt template can be found in Appendix D.4.

Technical Documentations. Given that users frequently ask package-related questions, we developed an additional subset called **Docs**, which transforms technical documentation from popular Python libraries—including *flask*, *pandas*, *pytorch*, *scikit*, and *seaborn*—into coding questions. When prompting LLMs to generate challenging yet clear and self-contained questions, we implemented a quality control mechanism allowing the model to abstain when the provided documentation proves insufficient for crafting high-quality coding questions. The complete prompt template is available in Appendix D.5.

More Questions. We further expand coding questions by employing MAGPIE (Xu et al., 2024) using seven open-source LLMs. We employ LLM annotators to classify generated questions, retaining only high-quality examples under "Algorithm Implementation" or "Function Generation" categories. We name this subset **Filter**. Details of this subset can be found in Appendix A. In addition, we synthesize more questions from existing *Package Instruct* (Huang et al., 2024) and *Evol Instruct* (Luo et al., 2023) synthetic datasets, and create two subsets named as **Package** and **Evol**.

Deduplication. After generating questions, we perform semantic deduplication within each subset by utilizing the `all-mpnet-base-v2` embedding model to project all questions into an embedding space. We then compute nearest-neighbor distances using FAISS (Douze et al., 2024), and filter out questions that surpass a predefined similarity threshold with existing entries.

2.2 Step 2: Solution & Test Generation

To generate verifiably correct coding solutions and unit tests $(sol, test)$ for questions from Step 1, we employ a **self-verification** procedure as detailed below. To ensure quality of solution and tests, we first employ *GPT-4o-0513* (which achieves state-of-the-art performance among non-reasoning models on the BigCodeBench Leaderboard (Zhuo et al., 2024)) to generate both solution and test, then execute these unit tests to validate the correctness of the solution. Only question-solution-test triplets that pass self-verification are retained. The complete prompt template is provided in Appendix D.6.

Since our goal is to generate verifiable training data that is challenging and diverse in coverage; however, even state-of-the-art models cannot guarantee bug-free code and ensure solutions pass their unit tests. Simply discarding questions that fail self-verification risks eliminating many challenging ones, potentially biasing our dataset towards a distribution of simple problems.

Our solution approach to address this challenge is the allocation of additional self-verification attempts for hard questions. For each question from Step 1, we allow up to a maximum of n attempts (where $n = 10$ in our experiments) to generate a solution that passes its unit tests. Importantly, each attempt fixes the question and regenerates both the solution and its corresponding unit tests from scratch. This approach preserves challenging questions while naturally assigning difficulty labels based on the success rate across attempts. Questions that fail to generate correct solutions after n attempts are discarded as they likely contain inherent flaws. Upon completing Step 2, we obtain a collection of 279K verified triplets.

2.3 Step 3: Post-training Data Synthesis

In creating LLM post-training data for coding tasks, there is a gap between coding questions (e.g., LeetCode) and training data. While coding questions are primarily expressed in natural language, training data needs to accommodate non-natural-language formats such as function calls and tool interactions. To address this disparity, we propose an LLM-based style converter to enhance the diversity of question formats. Specifically, we reformat each question q by taking its solution and test as inputs, $q' = \text{LLM}(q, sol, test)$, structuring them as Python completion tasks with function signatures and examples. Each reformatted question is paired with its original solution and test cases to form new triplets $(q', sol, test)$. This process results in the creation of 168K additional triplets, increasing our total to 447K, which are readily available for RL training.

Motivated by recent advances in reasoning models (Team, 2025; Labs, 2025), we further generate an SFT dataset for post-training by leveraging the questions in these triplets, using DeepSeek R1 (DeepSeek-AI et al., 2025) as the response generator to generate Chain-of-Thought responses. To ensure the quality of the generated responses, we generate 3 times for each question and perform test-based reject sampling, yielding a large-scale, high-quality, and verifiably correct SFT dataset for coding. We refer to this dataset as KodCode-SFT.

3 Analysis

In what follows, we conduct a comprehensive analysis to demonstrate the effectiveness of KodCode in generating diverse, challenging, and correct question-solution-test triplets.

3.1 Pipeline Analysis

Effectiveness of Self-Verification. To evaluate the reliability of our self-verification pipeline, we conduct experiments using the MBPP validation dataset (Austin et al., 2021), which contains 90 coding questions with ground-truth unit tests that were manually written and verified by humans. Following our pipeline, we generate both solutions and unit tests for these 90 questions. 10 questions are discarded as they fail all self-verification attempts, leaving us with 80 solutions that pass self-verification. We then evaluate these retained solutions against the MBPP ground-truth unit tests. We note that due to ambiguity in MBPP questions, some solutions that follow correct logic fail assertions due to mismatched input formats or numerical precision differences. After manual review of these edge cases, 78 out of 80 solutions pass all ground-truth unit tests (97.5% pass rate). This high success rate demonstrates our pipeline's effectiveness in generating verified solutions. Please refer to Appendix B.1 for analysis of the two failure cases.

Effectiveness of Allocating Additional Attempts to Challenging Questions. To assess the impact of allocating more attempts on solution & test generation, we adopt the $Pass@k$ metric commonly used in model evaluation literature (Austin et al., 2021; Chen et al., 2021). Specifically, we measure the proportion of questions for which at least one out of k solutions successfully passes its self-verification in Step 2.

The experimental results are illustrated in Figure 2, where we report $Pass@k$ by subsets and on average. We observe that while $Pass@1$ yields a low pass rate, increasing the number of trials from 1 to 5 results in an average pass rate increase of over 20%, and further increasing to 10 trials boosts the pass rate by an additional 4%. Notably, for more challenging tasks, such as Codeforces and Docs subsets, increasing the number of attempts significantly enhances pass rates. In contrast, simpler tasks like those in the Prefill subset show more modest gains from additional attempts. We emphasize that this scaling in attempts enables KodCode to retain more challenging questions that would otherwise be discarded.

3.2 Dataset Statistics and Analysis

In what follows, we analyze KodCode from the perspective of token length, diversity, difficulty, potential contamination, and data-flow analysis.

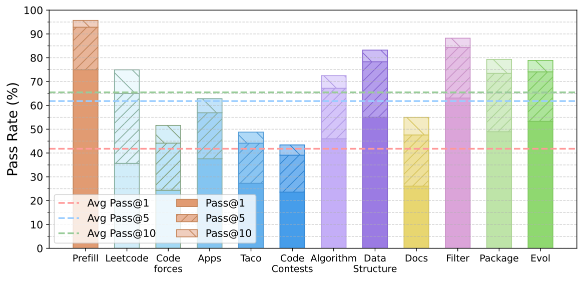

Figure 2: Statistics on pass rates via self-verification in Step 2 by subset with varing number of attempts.

Token Length and Unit Test Statistics. Figure 3 presents the distribution of token counts for questions and solutions across different subsets. We also provide unit test statistics of KodCode. Each coding question in KodCode contains an average of 7.52 unit tests.

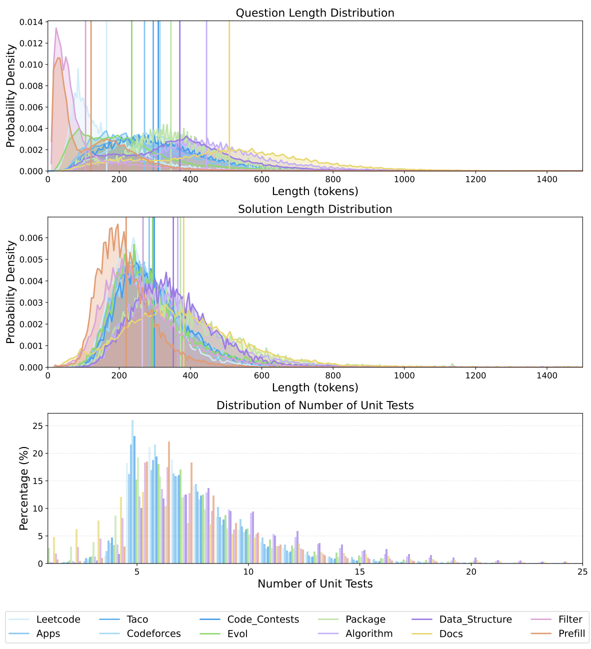

Figure 3: Distribution of token lengths for questions and responses, along with unit test counts across different subsets.

Diversity. We analyze the diversity of KodCode-V1's question distribution by comparing it with four baseline datasets: OSS Instruct (Wei et al., 2024c), ACECoder (Zeng et al., 2025), Educational Instruct (Huang et al., 2024), and Package Instruct (Huang et al., 2024). Using the `all-mpnet-base-v2` embedding model[*], we encode the questions and visualize their distribution using t-SNE (Van der Maaten and Hinton, 2008) to create a two-dimensional representation.

[*]https://huggingface.co/sentence-transformers/all-mpnet-base-v2

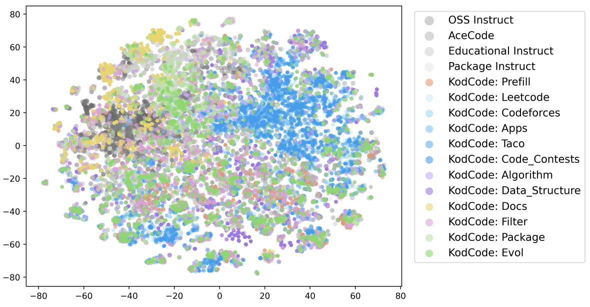

Figure 4: Comparison of t-SNE visualization between KodCode (by subset) and baseline datasets (OSS Instruct, ACECoder, Educational Instruct, and Package Instruct), with 2,000 sampled instructions per dataset.

The visualization in Figure 4 reveals two key observations. First, KodCode's question distribution (shown in color) spans the entire space, while baseline datasets (in gray) cluster primarily in the upper left region, demonstrating KodCode's broader topical diversity. Second, the Algorithm and Filter subsets of KodCode show comprehensive coverage across the entire space, validating their role in enhancing KodCode's overall diversity and robustness.

Difficulty. We analyze the difficulty distribution across KodCode subsets by examining the success rate of $(sol, test)$ pairs in self-verification across $n = 10$ attempts per question, as shown in Figure 5. We categorize questions into four difficulty levels: easy (pass rate >2/3), medium (1/3 to 2/3), hard (<1/3), and fail (all failures). The analysis reveals that Prefill tasks are typically the easiest, while Codeforces, Taco, and Code Contests subsets present the greatest challenges, as shown in their higher failure rates and larger proportions of hard questions.

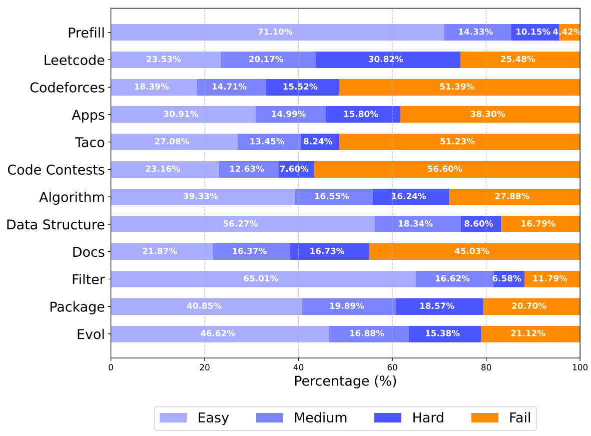

Figure 5: Difficulty distribution across subsets measured by pass rates.

Contamination Analysis. We evaluate potential contamination between KodCode and existing evaluation benchmarks, including HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), BigCodeBench (Zhuo et al., 2024), and LiveCodeBench (V5) (Jain et al., 2024). For each question in KodCode , we identify the most similar benchmark question using cosine similarity of embeddings, considering a question contaminated if the similarity exceeds 0.95.

As shown in Figure 6, our analysis reveals three findings. First, the contamination rate is minimal, with only 94 potentially contaminated questions out of 447K. Second, most overlaps occur between the Prefill subset and simple Python questions from HumanEval/MBPP. Third, the histogram in Figure 6-b shows that Prefill has the highest average maximum cosine similarity, while Docs has the lowest. We provide examples of contaminated questions in Appendix B.2 and exclude these cases from our performance evaluation in Section 4.

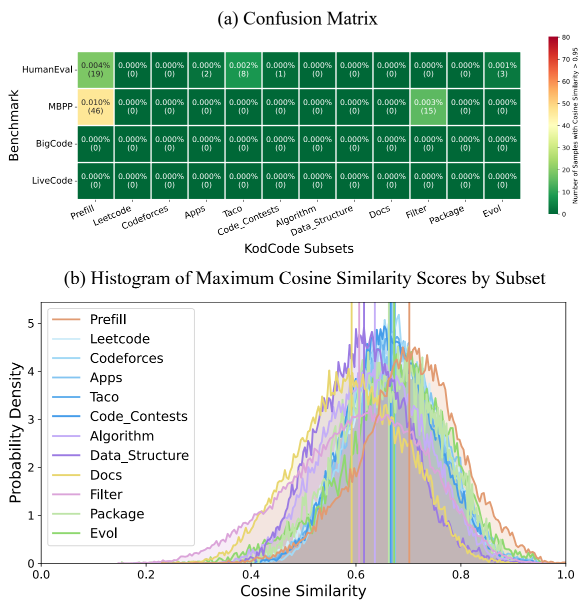

Figure 6: Contamination analysis between KodCode subsets and existing benchmarks. (a) Confusion matrix showing the percentage and absolute number (in parentheses) of contaminated samples with cosine similarity > 0.95. (b) Distribution of maximum cosine similarity scores across different KodCode subsets, with horizontal lines indicating subset averages.

Data Flow Analysis. Figure 7 presents a Sankey diagram illustrating data flow throughout our synthesis dataset generation pipeline. Over 25% of instances are eliminated during Step 1's deduplication process, with the Prefill subset showing the highest redundancy rate (over 50% discarded). In Step 2, instances failing unit tests are filtered out, with higher difficulty subsets (such as Codeforces and Taco) showing higher rejection rates.

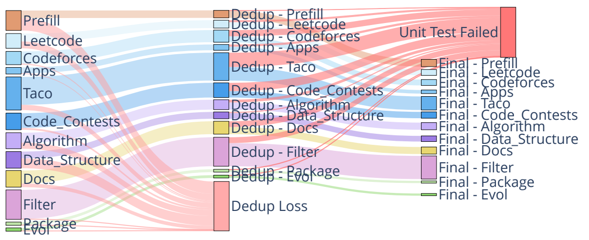

Figure 7: Data flow visualization through our pipeline: from initial subsets (left) through deduplication (middle) to final filtered sets after reject sampling (right). Red paths indicate discarded instances.

4 Performance Evaluation

To evaluate the performance of the KodCode dataset, we conduct supervised fine-tuning using Qwen2.5-Coder-32B-Instruct (Hui et al., 2024) on the KodCode-SFT dataset. We then compare the model's performance across several widely used code generation benchmarks against existing model baselines.

4.1 Experimental Setup

KodCode Setup. We evaluate two variants of KodCode-SFT. For preprocessing, we exclude R1 responses that are too long, too short, or implement class-based solutions rather than functional implementations. The first version, KodCode-50K, contains 50K instruction-response pairs selected based on the empirical results of mixing data from different sources and difficulties. The second version, KodCode-Hard-18K, contains all 18K instructions that are labeled as hard after applying the same preprocessing filters. We refer to the models fine-tuned on these two datasets as KodCode-32B-50K, and KodCode-32B-18K-Hard.

For SFT, we use a cosine learning rate schedule with a maximum learning rate of 1×10^{-5} when fine-tuning the Qwen-2.5 model. The maximum sequence length is 16384. The detailed training configurations can be found in Appendix C.1.

Baselines. We compare our KodCode-tuned model against several strong baselines, including non-reasoning models (Llama-3.3-70B-Instruct (Dubey et al., 2024), Llama-3.1-Tulu-3-70B (Lambert et al., 2025), Qwen-2.5-32B/72B-Instruct (Team, 2024a), Qwen-2.5-Coder-32B-Instruct (Hui

	Model Name	HumanEval		MBPP		BigCodeBench-C		BigCodeBench-I		LiveCodeBench (v5)			Average
		Base	Plus	Base	Plus	Full	Hard	Full	Hard	Easy	Medium	Hard	
Non-Reasoning Models	Llama-3.1-Tulu-3-70B	83.5	78.0	75.9	65.9	55.0	25.0	43.4	20.9	61.7	15.4	3.7	50.15
	Llama-3.3-70B-Instruct	82.9	77.4	87.3	73.0	57.9	29.1	47.0	26.4	81.4	21.1	8.5	55.5
	Qwen2.5-32B-Instruct	89.6	79.9	87.8	73.8	53.2	26.4	45.4	22.3	80.3	35.9	8.5	56.27
	Qwen2.5-72B-Instruct	87.8	81.1	90.2	76.2	57.5	33.1	46.1	21.6	69.9	39.9	7.0	57.15
	Qwen2.5-Coder-32B-Instruct	90.9	85.4	**90.2**	**77.0**	57.6	31.1	49.4	25.7	80.6	39.0	8.2	59.25
Reasoning Models	Sky-T1-32B-Preview	87.8	80.5	87.3	74.1	51.0	27.0	44.2	24.3	81.4	21.1	8.5	54.9
	QwQ-32B-Preview	87.8	82.3	84.4	69.8	53.9	26.4	38.8	23.0	90.0	51.7	10.0	56.75
	DeepSeek-R1-Distill-Qwen-32B	87.8	81.1	83.9	69.3	55.1	27.7	44.1	23.0	83.9	52.9	17.8	57.51
	DeepSeek-R1-Distill-Llama-70B	89.0	80.5	81.7	68.8	53.5	25.7	43.9	25.7	88.5	**56.2**	**18.9**	57.79
	Bespoke-Stratos-32B	88.4	83.5	88.1	75.1	56.2	33.1	47.3	27.0	86.7	49.5	10.4	59.64
KodCode	KodCode-32B-50K	**92.7**	85.4	89.9	76.2	**59.8**	**37.8**	**51.1**	**32.4**	87.8	35.9	6.7	61.22
	KodCode-32B-18K-Hard	90.9	**86.6**	89.2	**77.0**	59.7	37.2	50.5	31.1	**90.7**	39.3	5.6	**61.26**

Table 2: This table compares the model performances between KodCode-tuned models and strong baseline models across various benchmarks. Average scores are computed by first averaging sub-metrics within each benchmark and then taking the mean across all five benchmarks (HumanEval, MBPP, BCB-Complete, BCB-Instruct, and LiveCodeBench). Bold numbers indicate the best performance for each metric. KodCode-tuned models outperform larger baseline models, demonstrating the dataset's high quality and diversity.

et al., 2024)), and reasoning models (QwQ-32B-Preview (Team, 2024b), DeepSeek-R1-Distill-Llama-70B (DeepSeek-AI et al., 2025), Sky-T1-32B-Preview (Team, 2025), and Bespoke-Stratos-32B (Labs, 2025)).

Benchmarks and Evaluation Setups. We evaluate models on HumanEval(+) (Chen et al., 2021; Liu et al., 2024), MBPP(+) (Austin et al., 2021; Liu et al., 2024), BigCodeBench (Zhuo et al., 2024), and LiveCodeBench (V5) (Jain et al., 2024), each designed to assess different aspects of code generation, including functional correctness, external library usage, and competitive programming challenges. We use EvalPlus for HumanEval(+) and MBPP(+) evaluation, and Skythought-Evals (Li et al., 2025) for LiveCodeBench evaluation. We evaluate performance on both Complete and Instruct subsets of BigCodeBench.

We follow the official setups in each benchmark and evaluate all models using greedy decoding with a maximum generation length of 16,384 tokens. We follow the official chat templates of the model during inference.

4.2 Experimental Results

Model fine-tuned using KodCode-SFT outperforms baseline models. Table 2 presents a comparative analysis of Qwen-2.5-32B-Coder-Instruct fine-tuned on KodCode-SFT against various baseline models. Our fine-tuned models consistently outperform all baselines, including larger models, across both the Complete and Instruct subsets of BigCodeBench. Specifically, for BigCodeBench-C, our model achieves 59.8% (+1.9%) in the Full category and 37.8% (+4.7%) in the Hard category, compared to the strongest baseline. For BigCodeBench-I, it achieves 51.1% (+1.7%) in the Full category and 32.4% (+5.4%) in the Hard category, demonstrating strong performance across different evaluation settings. Additionally, on HumanEval, our model reaches 92.7%, surpassing Qwen2.5-Coder-32B-Instruct (90.9%) by 1.8%. For LiveCodeBench-Easy, our model attains 90.7%, exceeding all baselines, including QwQ-32B-Preview (90.0%). Overall, our model achieves an average score of 61.26%, the highest among all evaluated models, highlighting the quality and diversity of our KodCode dataset and its effectiveness in improving code generation performance across various benchmarks.

Effectiveness of hard coding questions. To validate the impact of challenging instances in KodCode, we compare models trained on two different 10K sample sets: one randomly sampled from the KodCode-50K dataset (named as KodCode-10K) and another from the KodCode-Hard-18K dataset (named as KodCode-Hard-10K). As shown in Table 3, KodCode-Hard-10K outperforms KodCode-10K on BigCodeBench-I Hard (31.8% vs. 27.7%, +4.1%) and BigCodeBench-I Full (50.6% vs. 49.9%, +0.7%), confirming that exposure to difficult coding problems improves model robustness. Similarly, for BigCodeBench-C Hard, KodCode-Hard-10K achieves 39.9%, surpassing KodCode-10K (38.5%, +1.4%). On LiveCodeBench Hard, KodCode-Hard 10K leads (6.3% vs. 4.8%, +1.5%), reinforcing that hard samples do enhance performance on complex programming tasks.

Benchmarks	KodCode Hard-10K	KodCode -10K	KodCode NoConvert-10K
BigCodeBench-C Full (%)	60.4	**61.1**	60.3
BigCodeBench-C Hard (%)	**39.9**	38.5	35.1
BigCodeBench-I Full (%)	**50.6**	49.9	49.6
BigCodeBench-I Hard (%)	**31.8**	27.7	28.4
LiveCodeBench Easy	**87.8**	**87.8**	86.4
LiveCodeBench Medium	**35.3**	32.6	32.6
LiveCodeBench Hard	**6.3**	4.8	5.6

Table 3: Ablation study of data selection when fine-tuning Qwen2.5-Coder-32B-Instruct on KodCode-SFT. Each model is trained on 10K sampled data. Bold numbers indicate best performance for each metric.

Effectiveness of style converter in KodCode generation process. To assess the impact of the style converter, we remove all instances processed by the style converter from the KodCode-50K dataset and randomly sample 10K instances from the remaining datasets for fine-tuning, naming the resulting dataset KodCode-NoConvert-10K. As shown in Table 3, KodCode-NoConvert-10K performs lower on BigCodeBench-C Full (60.3% vs. 61.1%) and BigCodeBench-C Hard (35.1% vs. 38.5%), indicating that removing style variations slightly reduces performance. In LiveCodeBench Easy, KodCode-NoConvert-10K scores 86.4%, lower than KodCode-10K (87.8%). This result is consistent with findings from (Li et al., 2025), highlighting that question format plays a role in code LLM performance.

5 Related Work

Synthetic Data Generation for Code LLMs. High-quality training data is crucial for LLM post-training (Zhou et al., 2023; Taori et al., 2023). Given the time and resource cost of human data collection (Databricks, 2023), synthetic data generation has emerged as a promising alternative. This approach leverages LLMs to produce synthetic instructions by expanding a small set of human-annotated seed instructions through few-shot prompting (Wang et al., 2023; Taori et al., 2023; Xu et al., 2023a,b; Wang et al., 2024; Sun et al., 2023). While synthetic data generation has been widely explored for alignment (Xu et al., 2024, 2023a; Ding et al., 2023; Cui et al., 2023) and mathematics (LI et al., 2024; Yue et al., 2024; Toshniwal et al., 2024) with millions of instances available, high-quality synthetic coding datasets remain scarce. Recently, several open-source coding datasets have been proposed by the community, including Code Alpaca (Chaudhary, 2023), OSS Instruct (Wei et al., 2024c), Evol Instruct (Luo et al., 2023), and Package Instruct (Huang et al., 2024). However, these resources are still limited in terms of diversity, difficulty, and scale. We present a comprehensive comparison between KodCode and existing open-source coding datasets in Table 1.

Code Generation with Execution Feedback. Ensuring the correctness of code generated by LLMs remains a critical challenge. Yang et al. (2024c) explores execution feedback within Docker environments, enabling models to rectify syntactic and logical errors by iteratively refining their outputs based on runtime execution results. Zheng et al. (2025) integrates code generation with execution and refinement to improve the quality of generated code.

LLM-based Unit Test Generation. While execution feedback helps identify code issues, unit tests play a complementary role by proactively assessing code correctness. Yang et al. (Yang et al., 2024d) provide a comprehensive empirical analysis on LLMs' capabilities in unit test generation. EvalPlus (Liu et al., 2024) enhances code evaluation by combining LLM-generated test cases with mutation-based expansion, increasing test diversity and rigor. Huang et al. (Huang et al., 2023) propose a multi-perspective self-consistency framework to select optimal code solutions based on self-generated tests. Recently, Wei et al. (Wei et al., 2024b) introduced an approach to align a base code model by generating solutions and verifying their correctness through self-generated unit tests. OpenCoder (Huang et al., 2024) employs a teacher model to generate multiple test cases for each code snippet, executing them in a Python interpreter and filtering out failing samples to ensure reliability. Jiao et al. (Jiao et al., 2025) enhance the reasoning capabilities of LLMs by evaluating solutions to reasoning problems via LLM-generated test case. AceCoder (Zeng et al., 2025) prompts GPT-4o-mini to generate unit tests, with Qwen 2.5 Coder-32B Instruct acting as a verifier to eliminate incorrect test cases.

6 Conclusion and Future Work

In this work, we presented KodCode, a large-scale synthetic dataset of 447K diverse coding questions paired with verified solutions and unit tests. Our three-step synthesis pipeline—comprising coding question generation, solution and test case refinement via self-verification, and post-training data synthesis—ensures both the diversity and quality of training data for high-performing coding language models. Through detailed analysis, we validate our pipeline's effectiveness and thoroughly examine the dataset's attributes. Comprehensive experiments demonstrate that models fine-tuned on KODCODEnot only achieve state-of-the-art performance across multiple benchmarks (HumanEval(+), MBPP(+), BigCodeBench, and LiveCodeBench) but also outperform larger models.

Future work will focus on three directions. First, we plan to scale up the dataset with more challenging problems, as our findings indicate that difficult instances significantly improve model performance. Second, we aim to investigate optimal strategies for post-training data selection. Finally, we will explore methods for generating repository-level synthetic data to further enhance coding LLMs.

Limitations

While models fine-tuned on KODCODE achieve state-of-the-art performance across most benchmarks, their performance on LiveCodeBench-Hard remains limited. This gap likely stems from insufficient representation of highly challenging competition-level programming problems in our dataset, which are prevalent in LiveCodeBench-Hard. Our current approach is also limited to using SFT to validate the quality of question–solution–test triplets. Future work could explore reinforcement learning techniques such as PPO (Schulman et al., 2017), GRPO (Shao et al., 2024), and REINFORCE++ (Hu, 2025) to further enhance model performance.

Ethical Statement

Our KODCODE dataset enhances code generation of code LLMs through diverse and challenging instruction-solution-test triplets. We do not introduce or endorse any applications that could cause harm or be misused. This paper does not present any ethical concerns.

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*.

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, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Ganqu Cui, Lifan Yuan, Ning Ding, Guanming Yao, Wei Zhu, Yuan Ni, Guotong Xie, Zhiyuan Liu, and Maosong Sun. 2023. Ultrafeedback: Boosting language models with high-quality feedback. *arXiv preprint arXiv:2310.01377*.

Databricks. 2023. Databricks dolly-15k.

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.

Ning Ding, Yulin Chen, Bokai Xu, Yujia Qin, Zhi Zheng, Shengding Hu, Zhiyuan Liu, Maosong Sun, and Bowen Zhou. 2023. Enhancing chat language models by scaling high-quality instructional conversations. *arXiv preprint arXiv:2305.14233*.

Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The faiss library.

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*.

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*.

Eric Hartford. 2023. LeetCode Solutions. Accessed: 2025-02-11.

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

Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *Preprint*, arXiv:2308.10620.

Jian Hu. 2025. Reinforce++: A simple and efficient approach for aligning large language models. *arXiv preprint arXiv:2501.03262*.

Baizhou Huang, Shuai Lu, Weizhu Chen, Xiaojun Wan, and Nan Duan. 2023. Enhancing large language models in coding through multi-perspective self-consistency. *arXiv preprint arXiv:2309.17272*.

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, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

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*.

Fangkai Jiao, Geyang Guo, Xingxing Zhang, Nancy F. Chen, Shafiq Joty, and Furu Wei. 2025. Preference optimization for reasoning with pseudo feedback. *Preprint*, arXiv:2411.16345.

Jur1cek. 2022. Codeforces Dataset. Accessed: 2025-02-11.

Keon. 2018. Pythonic Data Structures and Algorithms. Accessed: 2025-02-11.

Bespoke Labs. 2025. Bespoke-stratos-32b. https://huggingface.co/bespokelabs/Bespoke-Stratos-32B. Accessed: 2025-01-11.

Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V. Miranda, Alisa Liu, Nouha Dziri, Shane Lyu, Yuling Gu, Saumya Malik, Victoria Graf, Jena D. Hwang, Jiangjiang Yang, Ronan Le Bras, Oyvind Tafjord, Chris Wilhelm, Luca Soldaini, Noah A. Smith, Yizhong Wang, Pradeep Dasigi, and Hannaneh Hajishirzi. 2025. Tulu 3: Pushing frontiers in open language model post-training. *Preprint*, arXiv:2411.15124.

Bin Lei, Yi Zhang, Shan Zuo, and Caiwen Ding. 2024. Autocoder: Enhancing code large language model with aiev-instruct.

Dacheng Li, Shiyi Cao, Tyler Griggs, Shu Liu, Xiangxi Mo, Shishir G. Patil, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. 2025. Llms can easily learn to reason from demonstrations structure, not content, is what matters! *Preprint*, arXiv:2502.07374.

Jia LI, Edward Beeching, Lewis Tunstall, Ben Lipkin, Roman Soletskyi, Shengyi Costa Huang, Kashif Rasul, Longhui Yu, Albert Jiang, Ziju Shen, Zihan Qin, Bin Dong, Li Zhou, Yann Fleureau, Guillaume Lample, and Stanislas Polu. 2024. Numinamath. `[https://huggingface.co/AI-MO/NuminaMath-CoT](https://github.com/project-numina/aimo-progress-prize/blob/main/report/numina_dataset.pdf)`.

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

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel Mankowitz, Esme

Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *NeurIPS*, 36.

Lin Long, Rui Wang, Ruixuan Xiao, Junbo Zhao, Xiao Ding, Gang Chen, and Haobo Wang. 2024. On llms-driven synthetic data generation, curation, and evaluation: A survey. *arXiv preprint arXiv:2406.15126*.

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*.

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. *Preprint*, arXiv:2307.07924.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

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

Zhiqing Sun, Yikang Shen, Qinhong Zhou, Hongxin Zhang, Zhenfang Chen, David Cox, Yiming Yang, and Chuang Gan. 2023. Principle-driven self-alignment of language models from scratch with minimal human supervision. *Advances in Neural Information Processing Systems*, 36.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford alpaca: An instruction-following llama model. `https://github.com/tatsu-lab/stanford_alpaca`.

Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, et al. 2024. Gemma 2: Improving open language models at a practical size. *arXiv preprint arXiv:2408.00118*.

NovaSky Team. 2025. Sky-t1: Fully open-source reasoning model with o1-preview performance in $450 budget. https://novasky-ai.github.io/posts/sky-t1. Accessed: 2025-01-11.

Qwen Team. 2024a. Qwen2.5: A party of foundation models.

Qwen Team. 2024b. Qwq: Reflect deeply on the boundaries of the unknown.

The Algorithms. 2023. Python Algorithms. Accessed: 2025-02-11.

Shubham Toshniwal, Wei Du, Ivan Moshkov, Branislav Kisacanin, Alexan Ayrapetyan, and Igor Gitman. 2024. Openmathinstruct-2: Accelerating ai for math with massive open-source instruction data. *Preprint*, arXiv:2410.01560.

Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-sne. *Journal of machine learning research*, 9(11).

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.

Zifeng Wang, Chun-Liang Li, Vincent Perot, Long T Le, Jin Miao, Zizhao Zhang, Chen-Yu Lee, and Tomas Pfister. 2024. Codeclm: Aligning language models with tailored synthetic data. *arXiv preprint arXiv:2404.05875*.

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. *Preprint*, arXiv:2410.24198.

Yuxiang Wei, Federico Cassano, Jiawei Liu, Yifeng Ding, Naman Jain, Zachary Mueller, Harm de Vries, Leandro Von Werra, Arjun Guha, and Lingming Zhang. 2024b. Selfcodealign: Self-alignment for code generation. *arXiv preprint arXiv:2410.24198*.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024c. 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.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, and Daxin Jiang. 2023a. Wizardlm: Empowering large language models to follow complex instructions. *arXiv preprint arXiv:2304.12244*.

Canwen Xu, Daya Guo, Nan Duan, and Julian McAuley. 2023b. Baize: An open-source chat model with parameter-efficient tuning on self-chat data. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 6268–6278, Singapore. Association for Computational Linguistics.

Zhangchen Xu, Fengqing Jiang, Luyao Niu, Yuntian Deng, Radha Poovendran, Yejin Choi, and Bill Yuchen Lin. 2024. Magpie: Alignment data synthesis from scratch by prompting aligned llms with nothing. *arXiv preprint arXiv:2406.08464*.

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, Keqin Chen, Kexin Yang, Mei Li, Mingfeng 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, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zhihao Fan. 2024a. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*.

An Yang, Beichen Zhang, Binyuan Hui, Bofei Gao, Bowen Yu, Chengpeng Li, Dayiheng Liu, Jianhong Tu, Jingren Zhou, Junyang Lin, Keming Lu, Mingfeng Xue, Runji Lin, Tianyu Liu, Xingzhang Ren, and Zhenru Zhang. 2024b. Qwen2.5-math technical report: Toward mathematical expert model via self-improvement. *arXiv preprint arXiv:2409.12122*.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. 2024c. Intercode: Standardizing and benchmarking interactive coding with execution feedback. *NeurIPS*, 36.

Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, and Junjie Chen. 2024d. On the evaluation of large language models in unit test generation. *Preprint*, arXiv:2406.18181.

Xiang Yue, Tuney Zheng, Ge Zhang, and Wenhu Chen. 2024. Mammoth2: Scaling instructions from the web. *Preprint*, arXiv:2405.03548.

Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhu Chen. 2025. Acecoder: Acing coder rl via automated test-case synthesis. *ArXiv*, 2502.01718.

Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024a. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. *Preprint*, arXiv:2401.07339.

Quanjun Zhang, Chunrong Fang, Yang Xie, YuXiang Ma, Weisong Sun, Yun Yang, and Zhenyu Chen. 2024b. A systematic literature review on large language models for automated program repair. *Preprint*, arXiv:2405.01466.

Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. 2025. Opencodeinterpreter: Integrating code generation with execution and refinement. *Preprint*, arXiv:2402.14658.

Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Debug like a human: A large language model debugger via verifying runtime execution step-by-step. *Preprint*, arXiv:2402.16906.

Chunting Zhou, Pengfei Liu, Puxin Xu, Srini Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, LILI YU, Susan Zhang, Gargi Ghosh, Mike Lewis, Luke Zettlemoyer, and Omer Levy. 2023. LIMA: Less is more for alignment. In *Thirty-seventh Conference on Neural Information Processing Systems*.

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 Additional Information of the Filter Subset.

To create the Filter subset, we first generate data using seven state-of-the-art models: Llama-3.1/3.3-70B-Instruct (Dubey et al., 2024), Qwen2/2.5-72B-Instruct (Yang et al., 2024a; Team, 2024a), Qwen2.5-Coder-32B-Instruct (Hui et al., 2024), Qwen2.5-Math-72B-Instruct (Yang et al., 2024b), and Gemma-2-27b-it (Team et al., 2024). We then filter for Python-specific content in both instructions and responses, yielding 186K instances. Using *Llama-3.1-8B-Instruct* as our annotator, we label each instance for quality and difficulty (prompts provided in Appendix D.1). We retain only high-quality instances categorized as "Algorithm Implementation" or "Function Generation". Figure 8 shows the distribution of task categories before filtering. The final filtered dataset contains 89K coding questions. We follow Magpie's CC-BY-NC 4.0 license.

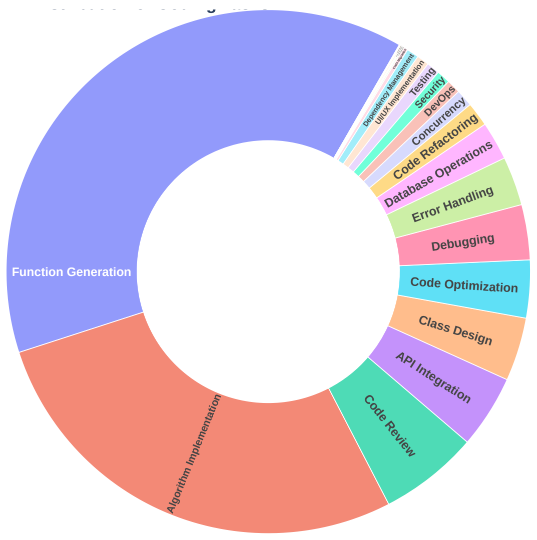

Figure 8: Task categories of the collected Magpie Coding data before filtering.

B More Analysis on KodCode

B.1 Analysis on MBPP Validation Failures Cases with Self Verification

Figures 9 and 10 illustrate two questions (Task 511 and Task 525) that failed in the unit test during our evaluation on the MBPP validation dataset. In Task 511, which requires computing the minimum sum of factors, the GPT-4o function attempts to minimize the sum of factor pairs by iterating up to the square root of the input number. However, it fails to correctly accumulate all prime factors when multiple factors exist beyond a single pair (e.g., 105, which has factors 3, 5, and 7), leading to incorrect results compared to the ground-truth (GT) implementation, which iteratively divides the number while summing all valid divisors.

In Task 525, the goal is to determine whether two given lines are parallel, with each line represented as a tuple of coefficients. The GT solution verifies parallelism by directly comparing slopes, $\frac{a_1}{b_1} = \frac{a_2}{b_2}$, ensuring compatibility with both the general case of two-element (a, b) tuples and three-element (a, b, c) representations. In contrast, GPT-4o applied an equivalent determinant-based condition, $a_1 \cdot b_2 = a_2 \cdot b_1$, but implicitly assumed that all inputs followed the three-element format, failing to account for the more general case where only (a, b) is provided. Due to this assumption, GPT-4o's implementation produced a mismatch in the third GT unit test case, where the input consists of two-element tuples $(3, 3)$ and $(5, 5)$, which implicitly represent valid lines with $c = 0$.

B.2 Examples of Dataset Contamination.

We present two examples of instances from KodCode that are similar to those in the MBPP and HumanEval benchmarks. These are illustrated in Figure 11 and Figure 12, respectively.

C More on Experimental Setups

C.1 Supervised Fine-Tuning Setups

Table 4 demonstrates the detailed supervised fine-tuning (SFT) hyper-parameters. We perform experiments on a cluster with 768 NVIDIA A100 GPUs. We fine-tune the model using 32 GPUs. The experiments in this paper were conducted using Llama Factory[†]. For datasets larger than 20K samples, we train for 2 epochs; for datasets smaller than 20K, we increase this to 3 epochs.

We adapt the system prompt from (Team, 2025), modifying it to better accommodate R1-style long chain-of-thought responses during fine-tuning. The full prompt can be found in Figure 13.

D Prompt Templates

D.1 Prompt Template for Dataset Labeling

Figure 14 and Figure 15 detail the prompts for labeling question quality and task category.

[†]https://github.com/hiyouga/LLaMA-Factory

Task_id: 511

Write a python function to find minimum sum of factors of a given number. The function name is find_Min_Sum.

GPT-4o Response

```python
def find_Min_Sum(n):
    """
    Returns the minimum sum of the factors
of a given number n.
    """
    if n <= 1:
        return n

    min_sum = float('inf')
    for i in range(1, int(n**0.5) + 1):
        if n % i == 0:
            j = n // i
            min_sum = min(min_sum, i + j)

    return min_sum
```

GT Response

```python
def find_Min_Sum(num):
    sum = 0
    i = 2
    while(i * i <= num):
        while(num % i == 0):
            sum += i
            num /= i
        i += 1
    sum += num
    return sum
```

GT Unit Test

```python
def test_find_Min_Sum():
    assert find_Min_Sum(12) == 7
    assert find_Min_Sum(105) == 15
    assert find_Min_Sum(2) == 2
```

Figure 9: Failed Task 511.

Task_id: 525

Write a python function to check whether two given lines are parallel or not. The function name is parallel_lines.

GPT-4o Response

```python
def parallel_lines(line1, line2):
    """
    Check if two lines are parallel.
    Each line is represented by a tuple (a, b, c) corresponding to
ax + by = c form.
    Two lines (a1, b1, c1) and (a2, b2, c2) are parallel if and
only if a1 * b2 == a2 * b1.

    Parameters:
    line1 (tuple): coefficients (a1, b1, c1) of the first line.
    line2 (tuple): coefficients (a2, b2, c2) of the second line.

    Returns:
    bool: True if the lines are parallel, False otherwise.
    """
    a1, b1, _ = line1
    a2, b2, _ = line2

    return a1 * b2 == a2 * b1
```

GT Response

```python
def parallel_lines(line1, line2):
    return line1[0] / line1[1] == line2[0] / line2[1]
```

GT Unit Test

```python
def test_parallel_lines():
    assert parallel_lines([2,3,4], [2,3,8]) == True
    assert parallel_lines([2,3,4], [4,-3,8]) == False
    assert parallel_lines([3,3],[5,5]) == True
```

Figure 10: Failed Task 525.

Example 1 of Contaminated Data

```
KodCode: How do I write a Python function to count the number of uppercase
    letters in a given string?

MBPP/450: Write a python function to count the upper case characters in a
    given string.

Cosine Similarity: 0.959
```

Figure 11: Example 1 of Contaminated Data

Example 2 of Contaminated Data

```
KodCode: def is_prime(n):
    """Returns True if n is a prime number, otherwise False.

    >>> is_prime(2) == True
    >>> is_prime(3) == True
    >>> is_prime(5) == True
    >>> is_prime(11) == True
    >>> is_prime(13) == True
    >>> is_prime(0) == False
    >>> is_prime(1) == False
    >>> is_prime(4) == False
    >>> is_prime(6) == False
    >>> is_prime(9) == False
    >>> is_prime(7919) == True
    >>> is_prime(8000) == False
    >>> is_prime(-1) == False
    >>> is_prime(-5) == False
    """

HumanEval/31: def is_prime(n):
    """Return true if a given number is prime, and false otherwise.
    >>> is_prime(6)
    False
    >>> is_prime(101)
    True
    >>> is_prime(11)
    True
    >>> is_prime(13441)
    True
    >>> is_prime(61)
    True
    >>> is_prime(4)
    False
    >>> is_prime(1)
    False

Cosine Similarity: 0.953
```

Figure 12: Example 2 of Contaminated Data

System Prompt for Fine-Tuning

Your role as an assistant involves thoroughly exploring questions through a systematic long thinking process before providing the final precise and accurate solutions. This requires engaging in a comprehensive cycle of analysis, summarizing, exploration, reassessment, reflection, backtracing, and iteration to develop well-considered thinking process. Please structure your response into two main sections: Thought and Solution. In the Thought section, detail your reasoning process using the specified format: <think> thought with steps </think>. Each step should include detailed considerations such as analisiying questions, summarizing relevant findings, brainstorming new ideas, verifying the accuracy of the current steps, refining any errors, and revisiting previous steps. In the Solution section after </think>, synthesize your analysis from the Thought section to present a clear, well-structured solution. Your final answer should be logically organized, precise, and include all essential implementation steps while maintaining conciseness.

Figure 13: System Prompt for Fine-Tuning

Prompt Template for Labeling Question Quality

```
# Instruction

You need to rate the quality of the code-related query based on its clarity, specificity,
    and completeness.

The rating scale is as follows:

- very poor: The query lacks critical code context (e.g., no code samples, error messages,
    or specific requirements). The problem description is vague or incoherent.
- poor: The query provides incomplete code context or unclear requirements. Important
    details like programming language, error messages, or expected behavior are missing.
- average: The query includes basic code context and requirements but may need clarification
     on specific behaviors, edge cases, or implementation details.
- good: The query provides clear code samples, specific requirements, and sufficient context
     (e.g., language version, environment, expected behavior). Minor details might be
    missing.
- excellent: The query is comprehensive with complete code examples, clear requirements,
    relevant error messages if applicable, expected behavior, and implementation constraints
    . All necessary context is provided for solving the problem.

If the query is not related to code, please rate it as `very poor`. If the query is short
    but clearly demonstrates the user's intent, please rate it as `good`.

If the query only contains code with no specific instructions, please rate it as `very poor`.


If the query explicitly asks to use programming language other than Python, please rate it
    as `very poor`.

## User Query
```
{input}
```

## Output Format
Given the user query, you first need to give an assesement, highlighting the strengths and/
    or weaknesses of the user query.
Then, you need to output a rating from very poor to excellent by filling in the placeholders
     in [...]:
```
{{
    "explanation": "[...]",
    "input_quality": "[very poor/poor/average/good/excellent]"
}}
```
```

Figure 14: Prompt Template for Labeling Question Quality.

Prompt Template for Labeling Task Category

~~~
# Instruction

Please label the task tags for the user query.

## User Query
```
{input}
```

## Tagging the user input
Please label the task tags for the user query. You will need to analyze the user query and
    select the most relevant task tag from the list below.

all_task_tags = [
    "Debugging",  # Users ask for help with debugging code. The user should provide the code
     and error message.
    "Code Review",  # Users ask for help with reviewing code. The user should provide the
    code.
    "Code Refactoring",  # Users ask for help with refactoring code to improve its structure.

    "Code Optimization",  # Users ask for help with improving code performance or efficiency.

    "Function Generation",  # Users ask for help with creating new functions based on
    requirements.
    "Class Design",  # Users ask for help with designing classes and object-oriented
    structures.
    "Algorithm Implementation",  # Users ask for help with implementing specific algorithms
    or data structures.
    "API Integration",  # Users ask for help with integrating third-party APIs or services.
    "Database Operations",  # Users ask for help with database queries, schema design, or
    operations.
    "Testing",  # Users ask for help with unit tests, integration tests, or test strategies.
    "Security",  # Users ask for help with security-related implementations or best
    practices.
    "Error Handling",  # Users ask for help with implementing error handling and validation.
    "Concurrency",  # Users ask for help with multi-threading, async programming, or
    parallel processing.
    "UI/UX Implementation",  # Users ask for help with frontend implementations or user
    interface code.
    "DevOps",  # Users ask for help with deployment, CI/CD, or infrastructure code.
    "Documentation",  # Users ask for help with code documentation or technical writing.
    "Dependency Management",  # Users ask for help with package management or dependency
    issues.
    "Code Migration",  # Users ask for help with migrating code between languages or
    frameworks.
    "Performance Profiling",  # Users ask for help with identifying and resolving
    performance bottlenecks.
    "Others"  # Any queries that do not fit into the above categories.
]

## Output Format:
Note that you can only select a single primary tag. Other applicable tags can be added to
    the list of other tags.
Now, please output your tags below in a json format by filling in the placeholders in <...>:
```
{{
    "primary_tag": "<primary tag>",
    "other_tags": ["<tag 1>", "<tag 2>", ... ]
}}
```
~~~

Figure 15: Prompt Template for Labeling Task Category.

Table 4: This table shows the hyper-parameters for supervised fine-tuning.

Hyper-parameter	**Value**
Learning Rate	1×10^{-5}
Number of Epochs	$2(> 20K)$ / $3(< 20K)$
Number of Devices	32
Per-device Batch Size	1
Gradient Accumulation Steps	4
Effective Batch Size	128
Optimizer	`Adamw`
Learning Rate Scheduler	`cosine`
Warmup Steps	100
Max Sequence Length	16384

D.2 Prompt Template for MAGPIE-Prefill

We utilize the following prompt template to query the *Qwen2.5-Coder-7B-Instruct* model. We implement the following three different prefillings and replace {Prefilling Content} in the template with one of the following options: (1) Write a function to, (2) Write a Python function, (3) Create a function that.

Prompt Template for MAGPIE-Prefill

```
<|im_start|>system
You are Qwen, created by Alibaba
    Cloud. You are a helpful
    assistant. You are designed to
     provide helpful, step-by-step
     guidance on coding problems.
    The user will ask you a wide
    range of coding questions.
Your purpose is to assist users in
     understanding coding concepts
    , working through code, and
    arriving at the correct
    solutions.<|im_end|>
<|im_start|>user
{Prefilling Content}
```

D.3 Prompt Template for Generating Coding Assessment Questions

Figure 17 demonstrates the prompt template for generating coding assessment questions.

D.4 Prompt Template for Converting DSA Code Snippets to Questions

Figure 18 demonstrates the prompt template for generating questions for the algorithm subset from the code & algorithm snippets.

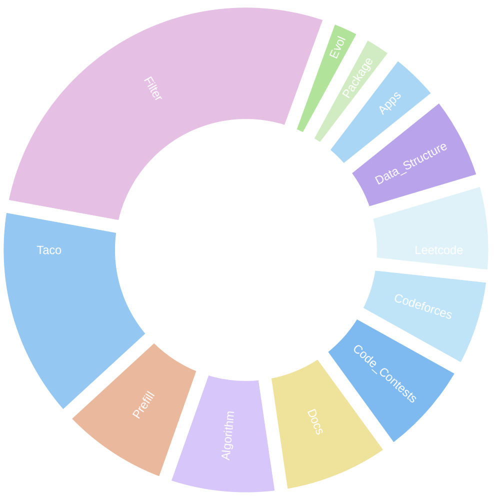

Figure 16: Subset Distribution of KODCODE.

D.5 Prompt Template for Converting Technical Documentations to Coding Questions

Figure 19 demonstrates the prompt template for generating questions for the Docs subset.

D.6 Prompt Template for Generating Solutions and Tests

Figure 20 demonstrates the prompt template for generating solutions and corresponding unit tests using GPT-4o-0513.

D.7 Prompt Template for Style Converter

Figure 21 demonstrates the prompt template for converting question styles.

E KODCODE Examples

We present representative examples from KODCODE subsets in Figure 22 to Figure 28. The distribution of each subset is demonstrated in Figure 16.

Prompt Template for Generating Coding Assessment Questions

```
## Task
Design a **Coding Assessment Question**.

## Objective
Analyze the provided sample questions and create an additional question that aligns with the
     existing set in terms of style, complexity, and scope.

## Guidelines

### Question Style
- Carefully examine the format and presentation of the given questions.
- Maintain a consistent tone and language used in the original set.

### Question Length
- Ensure your new question is of comparable length to the existing ones.
- If the original questions vary in length, aim for the average length among them.

### Difficulty Level
- Assess the cognitive and technical challenges presented in the sample questions.
- Match the complexity of concepts, algorithms, or programming techniques required.

### Topic Alignment
- Identify the core programming concepts or domains covered in the existing questions.
- Create a question that explores a related or complementary area within the same general
    topic.

### Question Uniqueness
- While maintaining similarity, ensure your new question is not a mere rephrasing of an
    existing one.
- Introduce a novel problem or scenario that tests the same skills in a different context.

## Output
Create a new question that matches the style of the existing set of questions. Output one
    new question only. Direct output the question, without adding question number or any
    other text in the beginning.

## Question 1
{Seed 1}

## Question 2
{Seed 2}

## Question 3
{Seed 3}

## Question 4
```

Figure 17: Prompt Template for Generating Coding Assessment Questions

Prompt Template for Algorithm Subset

```
## Task
Design a **Coding Assessment Question**.

## Objective
As an experienced programming instructor, you are designing programming questions to assess
    students' understanding of algorithms and data structures provided in the code snippets.
     Your question should require students to write code that demonstrates their
    comprehension of fundamental and advanced concepts from the code snippets.

## Guidelines

### Algorithm Analysis
Provide a thorough analysis of the algorithm or data structure provided in the code snippets,
     including but not limited to the following aspects:

#### Core Identification
* **Algorithm/Data Structure**: State the name, type (e.g., sorting algorithm, tree data
    structure), and main purpose.
* **Complexity**: Outline time and space complexity.
* **Principles**: Summarize its core operational steps or key mechanisms.

#### Characteristics & Applications
* **Properties**: Highlight essential properties (e.g., sorting stability, traversal order).
* **Common Use Cases**: Describe typical scenarios where this algorithm or structure is
    effective.
* **Strengths/Limitations**: Identify its key advantages and drawbacks, and specify when it'
    s most suitable to use or avoid.

#### Implementation Challenges
* **Edge Cases**: Describe common edge cases students should consider.
* **Performance Bottlenecks**: Identify any parts that can slow down execution or use excess
     memory.
* **Error Scenarios**: Explain situations that might lead to incorrect results if not
    handled correctly.
* **Optimization Points**: Note potential improvements or alternatives to enhance
    performance.

### Question Style
Based on the above analysis, craft a question that is:
* **Challenging** and requires a well-thought-out coding solution.
* **Clear and self-contained**, with all necessary information for solving it provided.
* **Focused on function implementation**, specifying:
  * Expected **input and output formats**.
  * Any **constraints or limitations**.
  * **Performance requirements**, if applicable.
* Enriched with a brief **scenario or context** (if it enhances clarity).
* Thoroughly examining the algorithm or data structure without referencing any example code
    directly.

## Output Format
Please use the following output format for consistency.

<|Analysis Begin|>

[Write your analysis here]

<|Analysis End|>

<|Question Begin|>

[Write the coding question here]

<|Question End|>

## Code Snippets
```

Figure 18: Prompt Template for Algorithm Subset

Prompt Template for Docs Subset

```
## Task
Design a **Coding Assessment Question**.

## Objective
As an experienced programming instructor, you are designing programming questions to assess
    students' understanding of {PACKAGE_NAME}. Your question should require students to
    write code that demonstrates their comprehension of fundamental and advanced concepts of
     this package.

## Guidelines

I will provide you with the documentation of {PACKAGE_NAME}. It could be a jupyter notebook,
     txt, or rst file. Please use it to design the question.

You should first analyze the documentation provided and understand the design of the package.
     Then, craft a question that is:
* **Challenging** and requires a well-thought-out coding solution.
* **Clear and self-contained**, with all necessary information for solving it provided.
* **Focused on function implementation**, specifying:
  * Expected **input and output formats**.
  * Any **constraints or limitations**.
  * **Performance requirements**, if applicable.

## Output Format
Please use the following output format for consistency.

<|Analysis Begin|>

[Write your analysis here]

<|Analysis End|>

<|Question Begin|>

[Write the coding question here. If you believe this document is not enough to design a
    question, please output "BAD_DOCUMENT" in this section.]

<|Question End|>

## Documentation
{CONTENT}

----------------
Now, please output the analysis and the question.
```

Figure 19: Prompt Template for Docs Subset

Prompt Template for Generating Solutions and Tests

~~~
## Task:
Please Answer the question and generate unit tests to verify your answer.

## Output Format:
Your solution and unit tests should be presented in markdown Python code format within the
    specified sections below. Ensure your code is within code blocks. For the tests, use
    pytest style by defining individual test functions (without classes) and using assert
    statements.

<|Solution Begin|>
[Solution Code in Python]
<|Solution End|>
<|Test Begin|>
[Unit Test Code in Python]
<|Test End|>

## Example
Below is an example output format implementing a simple a + b function.

<|Solution Begin|>
```python
def add(a, b):
    """
    Returns the sum of a and b.
    """
    return a + b
```
<|Solution End|>

<|Test Begin|>
```python
from solution import add

def test_add_positive_numbers():
    assert add(2, 3) == 5

def test_add_with_zero():
    assert add(0, 5) == 5
    assert add(5, 0) == 5

def test_add_negative_numbers():
    assert add(-1, -1) == -2

def test_add_mixed_sign_numbers():
    assert add(-1, 3) == 2
```
<|Test End|>

## Question:
~~~

Figure 20: Prompt Template for Generating Solutions and Tests

Prompt Template for Style Converter

~~~
## Task:
Please convert the given coding task to a coding completion task.

## Instruction

I will give you a coding task, and you goal is to convert it to a completion task. For
    example, if the coding task is "Find the longest common prefix among a list of strings
    .", the completion should be a task requiring the user to write a function that takes a
    list of strings as input and returns the longest common prefix as shown in the example
    below. Please ensure that the completion task contains the function definition,
    necessary imports, description, and test cases (if applicable).

I will provide you a coding task, along with a unit test and a solution. You can refer to
    them for the test cases and the function definition, but do not put the solution in the
    completion task.

Coding Task: Find the longest common prefix among a list of strings.

Unit Test:
```python
from solution import longest_common_prefix

def test_longest_common_prefix():
    assert longest_common_prefix(["flower", "flow", "flight"]) == "fl"
    assert longest_common_prefix(["dog", "racecar", "car"]) == ""
~~~

Solution:
```python
def longestCommonPrefix(strs: List[str]) -> str:
    # Sort the list of strings
    strs.sort()

    # Only need to compare first and last strings after sorting
    first = strs[0]
    last = strs[-1]

    # Find the common prefix between first and last
    i = 0
    while i < min(len(first), len(last)) and first[i] == last[i]:
        i += 1

    return first[:i]
```

Completion Task you should generate:
```python
def longest_common_prefix(strs: List[str]) -> str:
    """ Find the longest common prefix among a list of strings.
    >>> longest_common_prefix(["flower", "flow", "flight"]) "fl"
    >>> longest_common_prefix(["dog", "racecar", "car"]) ""
    """
```

Now, I will give you a coding task, and you goal is to convert it to a completion task.
## Input Information

Coding Task: [Coding Task Placeholder]

Unit Test: [Unit Test Placeholder]

Solution: [Solution Placeholder]

## Output Format:
<|Completion Begin|>
[Completion Task in Python]
<|Completion End|>

Please output the completion task strictly in the provided format, without adding any other
    information.
```

Figure 21: Prompt Template for Style Converter
```

**KODCODE Subset: Prefill**

```
Write a function to find max and min from an array in Python. I am looking
    for an O(n) time complexity solution.
```

Figure 22: Example of KODCODE Subset: Prefill

KODCODE Subset: Package

~~~
Design and implement a function that, given a list of integers, returns a new
     list where each element is the product of all other elements in the list
     except the one at the current index. You should solve this without using
     division and in O(n) time complexity.

The function should output:
- list: A list where each element is the product of all other elements in the
     input list except the element at the current index.

You should write self-contained code starting with:
```
def product_except_self(nums):
```
The first example from KodCode: Package: You are tasked with creating a
    function that calculates the similarity between two text documents based
    on the Jaccard similarity coefficient. The Jaccard similarity coefficient
     is defined as the size of the intersection divided by the size of the
    union of the sample sets. This function should take two strings as input,
     tokenize them into words, and then compute the similarity.

#### Function Specification:

**Function Name**: `jaccard_similarity`

**Parameters**:
- `doc1` (str): The first document as a string.
- `doc2` (str): The second document as a string.

**Behavior**:
1. Tokenize both input strings into sets of words.
2. Compute the intersection of these sets.
3. Compute the union of these sets.
4. Calculate the Jaccard similarity coefficient: (size of intersection) / (
    size of union).
5. Return the Jaccard similarity coefficient as a float.

**Example**:
```python
def jaccard_similarity(doc1, doc2):
    # Tokenize the documents
    set1 = set(doc1.split())
    set2 = set(doc2.split())

    # Compute intersection and union
    intersection = set1.intersection(set2)
    union = set1.union(set2)

    # Calculate and return Jaccard similarity coefficient
    return len(intersection) / len(union)

# Example usage
doc1 = "the cat in the hat"
doc2 = "the cat with a hat"
print(jaccard_similarity(doc1, doc2))  # Output: 0.5
```
~~~

Figure 23: Example of KODCODE Subset: Package

KodCode Subset: Codeforces

```
Many citizens of Gridland have taken up gardening as a hobby. Each gardener
    has a rectangular plot represented by a grid with $$$N$$$ rows and
    $$$M$$$ columns. Some cells in the grid may already contain flowers. Each
     gardener wants to plant saplings in the remaining empty cells such that
    no two saplings are adjacent to each other, neither horizontally,
    vertically, nor diagonally. The goal is to determine the maximum number
    of saplings that can be planted in the given grid.

### Input
- The first line of input contains two integers, $$$N$$$ and $$$M$$$ ($$$1 \
    leq N, M \leq 1000$$$), representing the dimensions of the grid.
- The next $$$N$$$ lines each contain $$$M$$$ characters, representing the
    grid. Each character is either a 'F' (which means a flower is already
    planted in that cell) or an 'E' (which means the cell is empty).

### Output
- Output a single integer, the maximum number of saplings that can be planted
    .

### Example

#### Input
```
3 4
E E E E
E F E E
E E E E
```

Output
```
4
```

Explanation
The optimal arrangement of planting saplings would be:

```
S E S E
E F E E
S E S E
```

Placing saplings ('S') in this way ensures no two saplings are adjacent and
    maximizes the number of saplings planted, which totals to 4 in this
    example.
```

Figure 24: Example of KodCode Subset: Codeforces

**KodCode Subset: Leetcode**

```
Given a list of integers `nums`, find the maximum product of any two distinct
     elements in the list. Return the maximum product. For example, given `
    nums = [5, 3, -1, 9, -7]`, the maximum product would be `5 * 9 = 45`.
```

Figure 25: Example of KodCode Subset: Leetcode

KODCODE Subset: Apps

~~~
## Task
Design a **Matrix Multiplication Function**.

## Objective
Write a function that multiplies two matrices of compatible dimensions and
    returns the result. Your implementation should include a check for
    dimension compatibility before attempting to multiply the matrices.

## Guidelines

### Function Signature
```python
def matrix_multiply(A, B):
    pass
~~~

### Input
- `A`: A list of lists representing matrix A, where each inner list
    represents a row of the matrix.
- `B`: A list of lists representing matrix B, where each inner list
    represents a row of the matrix.

### Output
- A new matrix representing the product of matrices A and B.

### Constraints
- The number of columns in matrix A should be equal to the number of rows in
    matrix B to be compatible for multiplication.
- Both matrices can be empty, and the function should return an empty matrix
    in such cases.

### Example
```python
A = [
    [1, 2, 3],
    [4, 5, 6]
]

B = [
    [7, 8],
    [9, 10],
    [11, 12]
]

print(matrix_multiply(A, B))
# Output: [[58, 64], [139, 154]]
```

### Explanation
The product of matrices A and B results in a new matrix where each element at
     position (i, j) is calculated as the sum of element-wise products of row
     i from matrix A and column j from matrix B.

### Constraints
1. Matrices A and B are rectangular.
2. Elements of matrices A and B can be any real numbers.

### Notes
- If the dimensions of the two matrices are incompatible, raise a ValueError
    with a message "Incompatible dimensions for matrix multiplication".
- The function should handle matrices with any number of rows and columns,
    but keep in mind the complexity of your solution, as matrix
    multiplication can be time-consuming for very large matrices.
```

Figure 26: Example of KODCODE Subset: Apps
```

KodCode Subset: Algorithm

~~~
You have been hired to develop a real-time weather dashboard that fetches and
     displays current weather data for a given list of cities. Your task is
    to write a function that processes and displays weather data from the
    OpenWeatherMap API for a specified list of cities. You need to ensure
    robust handling of potential issues such as missing or malformed data.
    You will implement the following function:

```python
def get_and_display_weather(cities: list) -> None:
    """
    Fetches current weather data from the OpenWeatherMap API for a specified
    list of cities and displays it in a formatted table.

    The API endpoint to use is:
    "http://api.openweathermap.org/data/2.5/weather?q={city}&appid={API_KEY}"

    Requirements:
    - Handle missing or malformed data gracefully, ensuring the program does
    not crash.
    - Display weather information in degrees Celsius.

    The table should display the following columns:
    - City
    - Weather (e.g., clear sky, rain)
    - Temperature
    - Humidity (%)
    - Wind Speed (m/s)

    Example of the table format:
    ```
    +-------------+------------+-------------+-----------+------------+
    | City        | Weather    | - Temperature
 | Humidity (%) | Wind Speed (m/s) |
    +-------------+------------+-------------+-----------+------------+
    | New York    | Clear sky  | 22          | 55        | 5          |
    | Los Angeles | Cloudy     | 18          | 60        | 3          |
    | ...                                                            |
    +-------------+------------+-------------+-----------+------------+
    ~~~

    Parameters:
    - cities: list of strings representing the city names.

    """
    pass
```

### Input/Output Format

- **Input**: A list of city names (e.g., ["New York", "Los Angeles", "Mumbai
    "]).
- **Output**: The function should output the formatted table with the columns
     specified above.

### Constraints

- Do not assume the API will always return all required fields for each city.
- Consider edge cases where some fields might be missing or improperly
    formatted.
- Ensure that the temperature conversion to Celsius is accurate.

### Performance Requirements

- Efficient handling of network requests and JSON parsing.
- Ensure the table can handle multiple cities without significant delay in
    processing time.
```

Figure 27: Example of KodCode Subset: Algorithm

KodCode Subset: Filter

```
Given a string, find the longest palindromic subsequence in that string.

Example:
Input: "banana"
Output: "anana"

Example:
Input: "abcd"
Output: "" (because there's no palindromic subsequence)

Note: A subsequence is a sequence that appears in the same relative order,
    but not necessarily contiguous.
```

Figure 28: Example of KodCode Subset: Filter