# OpenCoder: The Open Cookbook for Top-Tier Code Large Language Models

**Anonymous ACL submission**


## Abstract

Code LLMs have been widely used in various domains, including code generation, logical reasoning, and agent systems. However, open-access code LLMs mostly only release weights, lacking key features such as reproducible data pipelines and transparent training protocols, which are crucial for advancing deeper and more reliable investigations. To address the gap, we introduce **OpenCoder**, a top-tier code LLM that not only achieves performance comparable to industrial leading models but also serves as an "**open cookbook**" for the research community. Unlike most prior efforts, we release not only model weights and inference code, but also the reproducible training data, complete data processing pipeline, rigorous experimental ablation results, and detailed training protocols for open scientific research. Our work identifies the key ingredients for building a top-tier code LLM are: language-specific filtering rules, file-level deduplication , high-quality synthetic data and two-stage supervised fine-tuning strategy. By offering high level of openness, we aim to broaden access to all aspects of a top-tier code LLM, with OpenCoder serving as both a powerful model and an open foundation to accelerate research, enabling reproducible advancements in code intelligence.


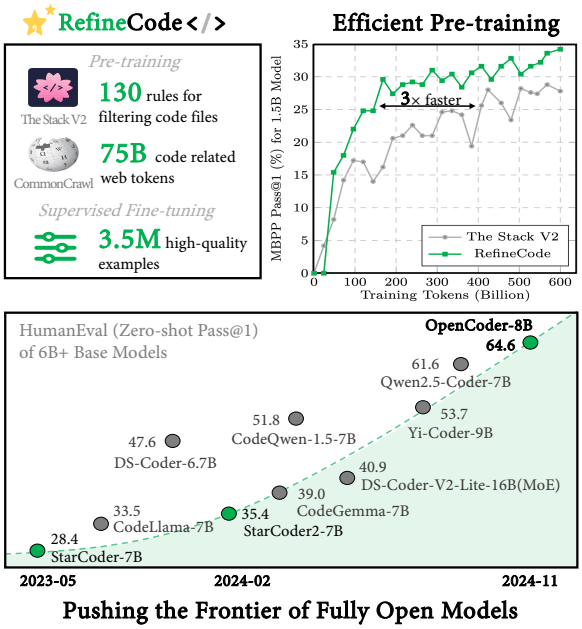


Figure 1: OpenCoder surpasses all previous fully open models (i.e., with open model weights and reproducible datasets) and other open-access models (i.e., with open model weights only) at the 6B+ parameter scale, pushing the frontier of fully open models to new heights.

## 1 Introduction

Large Language Models (LLMs) have achieved significant success in various domains (Wang et al., 2023; Que et al., 2024; Liu et al., 2024a,b; Wu et al., 2024), particularly in code-related tasks, revolutionizing the current paradigm of software development (Qian et al., 2024; Wang et al., 2024). Code-specific LLMs have emerged as a critical area within LLM research, with tools such as ChatGPT, Copilot, and Cursor reshaping the workflows of developers. Despite this, the performance of open-source LLMs focused on code (Li et al., 2023; Tao et al.; Lozhkov et al., 2024a) still falls short compared to state-of-the-art LLMs (Hui et al., 2024; Zhu et al., 2024), largely because these leading models keep their training datasets—an essential factor in LLM development—proprietary. This lack of transparency hinders the research community from establishing strong baselines and gaining deeper insights into top-tier code LLMs.

To remedy the gap, we set three primary goals by releasing **OpenCoder** and its development materials: (i) We aim to provide scholars with a meticulously curated and fully transparent strong baseline code LLM for research on mechanical interpretability and the data distribution of code LLMs. (ii) We intend to conduct in-depth investigations into the pretraining and instruction data curation pipeline

for the development of stronger code LLMs. (iii) By enabling a detailed review of the development of the models, we hope to unlock more diverse customized solutions based on transparent code LLM. Through OpenCoder, we strive to stimulate and accelerate the growth of the open-source code LLM community.

Our comprehensive set of controlled experiments highlights key design choices for data curation for advanced code LLMs in different training stages. **During pre-training Stage**: (i) Effective data cleaning is crucial (Zhou et al., 2024), requiring well-designed heuristic rules to process large-scale corpora under limited resources and visualization to perceive data distribution. (ii) The impact of deduplication is significant, with file-level deduplication proving to be more effective than repository-level deduplication by maintaining data diversity and enhancing model performance on downstream tasks (Li et al., 2023). (iii) The influence of GitHub stars is also examined, revealing that filtering data based on Github star count can possibly reduce data diversity and affect the overall data distribution, contributing to a suboptimal result (Allal et al., 2023). Moreover, **in the annealing phase**, high-quality data is crucial for further enhancing the model's capabilities, indicating that data quality is more important than quantity in the later stages of model training. Finally, **during instruction tuning phase**, a two-stage instruction tuning strategy allows the model to acquire broad capabilities initially and then refine them with code-specific tasks, resulting in improved performance on both theoretical and practical coding tasks.

Our contribution is summarized below:

- We present **OpenCoder**, a top-tier code llm archieving competitive performance with leading models across multiple benchmarks.
- We provide an full-stack **open cookbook** for code LLMs, including pipeline, training sets and middle checkpoints as detailed in Table 1.
- We identify the **key ingredients** for building a top-tier code LLM, including language-specific heuristic rules, file-level duplication, synthetic-data and two-stage SFT.

## 2 Pretraining Data

Pretraining data plays a crucial role in the development of LLMs, where the scale, quality, and diversity of the data greatly affect the model's overall performance. To this end, we present how to process massive datasets with fine-grained heuristic rules under limited computational resources, and analyze the overall data distribution through visualization. This section will comprehensively illustrate the data processing strategies used in the general pretraining stage and the annealing stage.

| Model | Pipe | PT-data | SFT-data | Mid-ckpts | Tokens | HE |
|---|---|---|---|---|---|---|
| *Open Model Weights & Reproducible Datasets* | | | | | | |
| OpenCoder-8B | ✓ | ✓ | ✓ | ✓ | 2.5 | 83 |
| StarCoder2-15B | ✓ | ✓ | ✗ | ✗ | 4.1 | 72 |
| Crystal-7B | ✗ | ✓ | ✗ | ✓ | 1.3 | 34 |
| *Open Model Weights* | | | | | | |
| CodeLlama-7B | ✗ | ✗ | ✗ | ✗ | 2.5 | 34 |
| CodeGemma-7B | ✗ | ✗ | ✗ | ✗ | 6.5 | 56 |
| DS-Coder-V2-Lite | ✗ | ✗ | ✗ | ✗ | 10.2 | 81 |
| Yi-Coder-9B | ✗ | ✗ | ✗ | ✗ | 6.0 | 85 |
| Qwen2.5-Coder-7B | ✗ | ✗ | ✗ | ✗ | 23.5 | 88 |

Table 1: Comparison of open-source resources among code LLMs. **Pipe**: pretraining data cleaning pipeline; **PT-data**: reproducible pretraining data; **SFT-data**: large-scale SFT corpus (>1M samples); **Mid-ckpt**: intermediate pretraining checkpoints; **Tokens**: total training tokens(B);**HE**: HumanEval scores for chat models.

### 2.1 RefineCode

Pretraining data forms the foundation for the capabilities of LLM. While The Stack v2 (Lozhkov et al., 2024a) has been a valuable resource for training code LLMs in the open-source community, its quality is insufficient for top-tier model performance.

To address this, we introduce **RefineCode**, a high-quality, reproducible dataset of 960 billion tokens across 607 programming languages, comprising raw code and code-related web data. RefineCode is the first pre-training code dataset that performs language-specific refinement, employing customized cleaning thresholds and domain-adapted rules for each programming language. A comparison between RefineCode and all versions of The Stack is provided in Table 2.

#### 2.1.1 Raw Code

We collect raw code primarily from GitHub repositories up to November 2023 and non-GitHub data from The Stack v2. To ensure the curation of high-quality raw code data, we have developed the code-specific data processing pipeline including modules

| | # Total | # Web | # Programs | # Rules | LS Rules |
|---|---|---|---|---|---|
| The Stack v1 | 200 B | \ | 88 | ~15 | ✗ |
| The Stack v2 | 900 B | ~30 B | 619 | ~15 | ✗ |
| **RefineCode** | 960 B | ~75 B | 607 | ~130 | ✓ |

Table 2: Comparison of training data between **RefineCode** and The Stack series. "Total" represents the total tokens, "Web" indicates tokens from web-related texts, "Programs" refers to programming languages, "Rules" denotes filtering rules applied, and "LS Rules" represents the language-specific filtering rules.

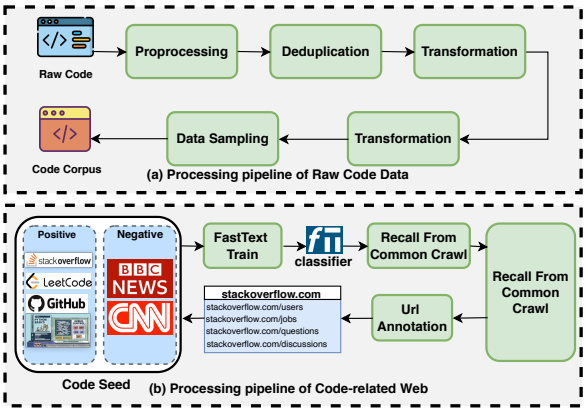


Figure 2: Illustration of RefineCode pipeline

of **preprocessing**, **deduplication**, **transformation**, **filtering**, and **data sampling**. We briefly outline the pipeline, with additional details provided in Appendix A.

**Preprocessing** To optimize resources, we exclude files larger than 8 MB, as they are typically non-text and resource-intensive. We then filter for files related to programming languages based on their extensions, as defined by *linguist* (Linguist, 2024), and discard those with low capacity or quality. This results in a final selection of 607 distinct programming language file types.

**Deduplication** Deduplication is a crucial module in the data pipeline to enhance both pretraining efficiency and efficacy (Lee et al., 2021). We first perform exact deduplication using SHA256 to eliminate fully duplicate files, then apply fuzzy deduplication. For the latter, we use MinHash (Broder, 1997) and LSH (Leskovec et al., 2014) to remove near-identical files.

**Transformation** To address minor issues without discarding entire files, we apply two transformation rules before filtering: (1) we remove repetitive

| Category | Source | # Tokens | Per. |
|---|---|---|---|
| Raw Code | Github | 755 B | 78.4% |
| | Jupyter Notes | 11 B | 1.1% |
| | The Stack v2 | 120 B | 12.5% |
| Code-related Web | CC | 13 B | 1.4% |
| | SkyPile | 3 B | 0.3% |
| | FineWeb | 55 B | 5.7% |
| OpenSource | AutoMathText | 3 B | 0.3% |

Table 3: The Composition of **RefineCode**.

and irrelevant copyright notices from the beginning of over 15% of code files; and (2) to mitigate privacy risks, we detect and replace Personally Identifiable Information (PII)—such as passwords and emails—with placeholders like "<name>" and "<password>" using regular expressions.

**Filtering** The quality of code files on GitHub varies significantly, with lower-quality code potentially hindering LLM pretraining. We propose the first heuristic filtering framework tailored to code pretraining data by considering the unique characteristics of different programming languages. This framework provides over 130 heuristic rules with customized weight assignments across three categories, resulting in more precise and higher-quality data cleansing. Detailed heuristic rules and high-level design principles are provided in Appendix B.

**Data Sampling** Structured data formats with specific syntax (e.g., JSON/HTML) present a unique challenge in code pretraining. While these formats represent a substantial portion of publicly available code corpora, their disproportionate representation in pre-training data may paradoxically undermine model generalization capacity through pattern memorization. Similar to modern code processing methods (Lozhkov et al., 2024a,b) we perform downsampling on programming languages (e.g., HTML, Java).

### 2.1.2 Code-Related Web Data

Inspired by DeepSeekMath (Shao et al., 2024), we collect high-quality code-related data corpus from the Common Crawl dataset, Fineweb (Penedo et al., 2024), skyPile (Wei et al., 2023a) and web part of AutoMathText Dataset (Zhang et al., 2024b). Due to the lack of open-source fine-gained code corpus, we first annotate 500k high-quality code-like data from CommonCrawl using the Autonomous Data Selection (Zhang et al., 2024b) method as seed.

### 2.2 Annealing Data

Following the training strategy in MiniCPM (Hu et al., 2024), our model undergoes a rapid learning rate annealing phase after the general pretraining stage, where very high-quality training data is used to further enhance the model's capabilities. In addition to the RefineCode from the original distribution, we further incorporated the Algorithmic Corpus and synthetic data during the annealing phase.

**RefineCode** During the annealing stage, we maintain distribution consistency with the pretraining phase to prevent catastrophic forgetting (Hu et al., 2024; Shen et al., 2024). 84% of the annealing data is drawn from the original RefineCode.

**Algorithmic Corpus** Algorithmic code files exhibit strong code logic and minimal dependency on external files, demonstrating excellent self-containment. They align well with the smaller, independent tasks typical of real-world interactive scenarios. Therefore, we extract a subset of the pretraining data containing keywords like "def solution" or "class solution" to construct this corpus. We also use model-based method, the results in section shows that rule is better.

**High Quality Code Snippet** Inspired by the synthetic CodeExercises dataset in Gunasekar et al. (2023), we utilized the algorithmic corpus as seeds and employed LLM to synthesize self-contained independent functions along with corresponding test cases. We only retained the data that successfully passed the test cases. We extend this pipeline to support multiple program languages.

**Code Textbook** Pretraining data with a clear semantic mapping between code and natural language is scarce. (Song et al., 2024). To address this issue, we utilize a powerful LLM to extract and elaborate on abstract code knowledge from high-quality datasets like HQCode (Yuxiang630, 2024). This approach is designed to help the model learn code from diverse perspectives.

### 2.3 Visual Inspection

The pretraining data processing pipeline (e.g., deduplication, filtering) involves numerous hyperparameters, making ablation studies for each economically infeasible. Instead, we use PCA to visualize embeddings extracted from CodeBERT (Feng et al., 2020) and perform spot checks on outliers, providing an effective way to understand the distribution of cleaned pretraining data.

| Category | Dataset | # Token |
|---|---|---|
| Original Data | RefineCode | 83.94 B |
| | Algorithmic Corpus | 12.44 B |
| Synthetic Data | High Quality Code Snippet | 2.71 B |
| | Code Textbooks | 0.91 B |

Table 4: Detailed data mixture for annealing data.

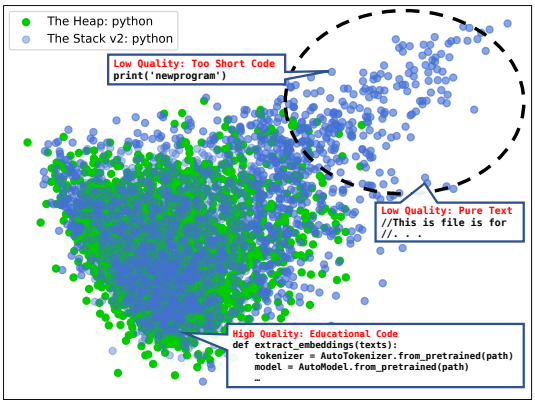


Figure 3: Visualization on the python of **RefineCode** and The Stack v2.

Interestingly, visualization reveals the quality gap between RefineCode and The Stack v2 even before pretraining. As shown in Figure 4, The Stack V2 data shows a greater number of outliers, while the embeddings of RefineCode appear more tightly clustered. Besides, after analyzing the outlier data, we observe the outliers usually show many low-quality patterns, such as pure text comments, hexadecimal-only data, and excessively short code lacking computational logic, which can distort the distribution of the pretraining dataset and ultimately hurt the efficiency of pretraining.

## 3 Pretraining

### 3.1 Model Architecture

OpenCoder follows the architecture of LLaMA 3 (Dubey et al., 2024) and is in two sizes: 1.5B and 8B parameters. The 1.5B model features 24 layers, a hidden size of 2240, and 14 attention heads, with a context window size of 4096. The 8B model has 32 layers, a hidden size of 4096, 32 attention heads, and grouped query attention with 8 key-value heads. Both models employ SwiGLU (Shazeer, 2020) and utilize the tokenizer proposed by INF-Team (2024). Detailed configurations are presented in Table 5.

| | 1.5B | 8B |
|---|---|---|
| Layers | 24 | 32 |
| Model Dimension | 2240 | 4096 |
| Attention Heads | 14 | 32 |
| Key / Value Heads | 14 | 8 |
| Activation Function | SwiGLU | |
| Vocab Size | 96640 | |
| Positional Embedding ($\theta$) | 10,000 | 500,000 |
| Context Window Size | 4096 | 8192 |

Table 5: Overview of the key hyperparameters of OpenCoder, including 1.5B and 8B. RoPE is selected for Positional Embedding.

### 3.2 Training Details

**Optimizer** Both models employ the WSD learning schedule (Hu et al., 2024). The schedule include a warm-up phase of 2,000 steps over 8B tokens, followed by a peak learning rate of 3e-4, which remained constant after the warm-up. During the final 100B token annealing phase, the learning rate decayed exponentially from 3e-4 to 1e-5.

**Training Framework** The training for both models was conducted using Megatron-LM (Shoeybi et al., 2020) with distributed optimization and DDP gradient overlap. The 1.5B model was trained on 2 trillion tokens with a sequence length of 4096, a micro-batch size of 4, and a global batch size of 1024. The training process was conducted on a cluster of 256 H800 GPUs over a duration of 109.5 hours, totaling 28,034 GPU hours. For the 8B model, training was performed on 2.5 trillion tokens with a sequence length of 8192, a micro-batch size of 1, tensor parallelism (TP) of 2, and a global batch size of 1024. This training was executed on 512 H100 GPUs over 187.5 hours, resulting in a total of 96,000 GPU hours.

## 4 Post Training

We constructed a diverse dataset of over 1 million instructions to fine-tune opencoder. In addition to open-source data, we employ multiple synthetic approaches to construct code instruction datasets. Full details are provided in appendix H.

### 4.1 Data Collection

**Open-source Training Data** We curate a collection of high-quality open-source code instruction datasets, including Evol-Instruct (Luo et al., 2024), Infinity-Instruct (BAAI, 2024), and McEval (Chai et al., 2024). Furthermore, we extract real user queries from WildChat (Zhao et al., 2024) and

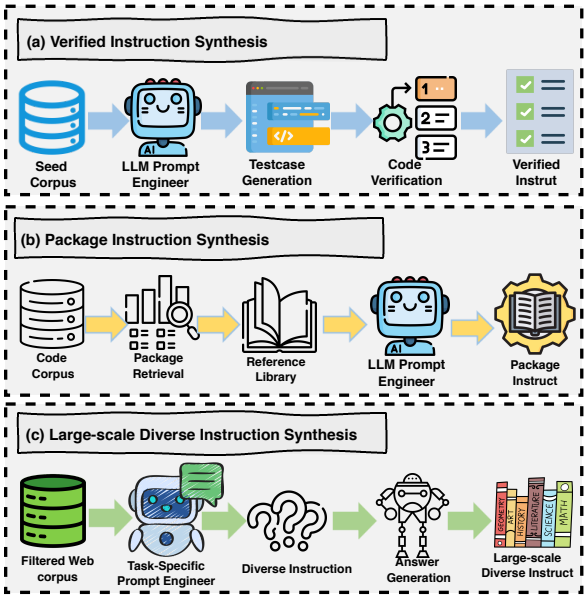


Figure 4: Illustration of synthetic instruction workflow

Code-290k-ShareGPT (Computations, 2023), and employ LLM to identify code-related dialogue histories, followed by rigorous data cleaning. The resulting dataset, termed RealUser-Instruct, not only demonstrates high diversity but also closely mirrors real-world problem complexity, aligning to authentic scenarios.

**Verified Instruction Synthesis** Following oss-instruct (Wei et al., 2023b), we utilizes raw code as initial seed to generate question-answer pairs. To further enhance code quality, we implement a rigorous verification process through test case execution. Our methodology involves: (1) sampling high-quality code from RefineCode; (2) using a teacher model to generate question-answer pairs with chain-of-thought reasoning and multiple test cases for code segments; (3) executing these test cases using code interpreter. (4) retain codes passing over 80% test executions.

**Package Instruction Synthesis** Package instruction data are generated to further enhance proficiency of code LLM with common packages. Our methodology involves a four-step process: (1) extracting high-quality code snippets using packages from RefineCode; (2) retrieving the corresponding API documentation and usage guidelines from PyDoc; (3) prompting a teacher model to generate QA data based on the snippets and documentation; (4) prompting a strong model to verify if the QA data adhere to the API usage guidelines.

**Large-scale Diverse Instruction Synthesis** Pre-training web corpus contains a vast and diverse collection of reasoning data (Yue et al., 2024). We develop a large-scale instruction framework comprising four key components: (1) Context refinement, where an LLM filters irrelevant web content and extracts meaningful sentences as question seeds; (2) Task specification, which defines programming languages, difficulty levels, and task types through a configurable module, with prompt engineering generating diverse, context-rich templates; (3) Content generation, where an advanced LLM produces both questions and corresponding answers, validated through automated code execution and unit testing; (4) Response refinement, where an LLM enhances outputs with code comments and detailed explanations.

### 4.2 Two-Stage Training Strategy

To develop a language model proficient in both theoretical and practical aspects of computer science, we implement a two-stage instruction fine-tuning process. In the first stage, we enhance the model's theoretical understanding with a comprehensive and diverse set of domain-specific question-answer pairs. This stage covers a wide range of topics, including algorithms, and data structures, enabling the model to provide accurate responses to complex theoretical queries that span both computer science theory and real-world user scenarios. The second stage is more focused, concentrating on practical downstream tasks by refining the model's code generation and error correction capabilities to ensure strong performance in real-world applications.

| Stage | Data Source | # Examples |
|---|---|---|
| Stage1 | RealUser-Instruct | 0.7 M |
| | Large-scale Instruct | 2.3 M |
| | Infinity-Instruct | 1.0 M |
| Stage2 | McEval-Instruct | 36 K |
| | Evol-Instruct | 111 K |
| | Educational-Instruct | 110 K |
| | Package-Instruct | 110 K |

Table 6: Detailed data composition of Two-Stage SFT

## 5 Experimental Results

This section contains evaluations led to OpenCoder with many more multilingual evaluation details provided in appendix G.

**Base Evaluation** In Table 7 we benchmark OpenCoder-base series on common downstream tasks. We find that OpenCoder series achieve the best performance among fully open models (green lines), pushing the frontier of open-source code llm. On the widely-used code benchmarks HumanEval(+) and MBPP(+), OpenCoder-base achieve state-of-the-art performance, surpassing leading industrial code llms.

**Chat Evaluation** In Table 8 we benchmark OpenCoder-instruct series on common tasks. OpenCoder-1.5B-instruct maintains leading performance among all 1.5B models. In BigCodeBench, a benchmark reflecting the comprehensive capabilities of code llms, OpenCoder shows strong performance. OpenCoder-8B was trained on a total of 2.5 T tokens, significantly fewer than industrial codellm (e.g., Yi-Coder-9B: 6.0T; Qwen2.5-Coder-7B: 23.5T). At 8B scale, OpenCoder's performance remains in the top tier and maintains a leading position among fully open-source models.

## 6 Ablation Study

**File-level deduplication outperforms repo-level deduplication for code corpus** Data deduplication improves efficiency and reduces overfitting, with both file-level and repo-level methods being applied for code data (Lee et al., 2021; Lozhkov et al., 2024a; Guo et al., 2024). We conduct a detailed comparison of deduplication levels, with experiments details in Appendix C and key findings listed below: (i) File-level deduplication leads to better training efficiency despite more aggressive token reduction. File-level dedup retains only a third of the tokens compared to repo-level deduplication but results in higher training efficiency, as shown in Table 9. (ii) Repo-level dedup leaves high redundancy. With further analysis of repo-level dedup results, we find that 52B tokens (52%) exhibits complete character-level equivalence with another file. When conducting file-level dedup as post-processing step, 68B tokens (68%) could be further deduplicated. The performance trending can be found in Figure 5(a).

**High-quality code data in annealing significantly boosts performance** We compared the impact of high-quality data (the Algorithmic Corpus and Synthetic Data) during the annealing phase. From Figure 5(b), we observe that the performance drops a lot when the high-quality training data is removed, which demonstrates the effectiveness of our constructed high-quality data in the annealing phase.

| Model | Size | HumanEval | | MBPP | | | BigCodeBench | |
|---|---|---|---|---|---|---|---|---|
| | | *HE* | *HE+* | *MBPP* | *MBPP+* | *3-shot* | *Full* | *Hard* |
| **1B+ Models** | | | | | | | | |
| DeepSeek-Coder-1.3B-Base | 1.3B | 34.8 | 26.8 | 55.6 | 46.9 | 46.2 | 26.1 | 3.4 |
| Yi-Coder-1.5B | 1.5B | 41.5 | 32.9 | 27.0 | 22.2 | 51.6 | 23.5 | 3.4 |
| CodeGemma-2B | 2B | 31.1 | 16.5 | 51.1 | 43.1 | 45.4 | 23.9 | 7.4 |
| Qwen2.5-Coder-1.5B | 1.5B | 43.9 | 36.6 | 69.2 | 58.6 | **59.2** | **34.6** | **9.5** |
| StarCoder2-3B | 3B | 31.7 | 27.4 | 60.2 | 49.1 | 46.4 | 21.4 | 4.7 |
| OpenCoder-1.5B-Base | 1.5B | **54.3** | **49.4** | **70.6** | **58.7** | 51.8 | 24.5 | 5.4 |
| **6B+ Models** | | | | | | | | |
| CodeLlama-7B | 7B | 33.5 | 26.2 | 55.3 | 46.8 | 41.4 | 28.7 | 5.4 |
| CodeGemma-7B | 7B | 39.0 | 32.3 | 50.5 | 40.7 | 55.0 | 38.3 | 10.1 |
| DS-Coder-6.7B-Base | 6.7B | 47.6 | 39.6 | 70.2 | 56.6 | 60.6 | 41.1 | 11.5 |
| DS-Coder-V2-Lite-Base (MoE) | 16B | 40.9 | 34.1 | 71.9 | 59.4 | 62.6 | 30.6 | 8.1 |
| CodeQwen1.5-7B | 7B | 51.8 | 45.7 | 72.2 | 60.2 | 61.8 | 45.6 | 15.6 |
| Yi-Coder-9B | 9B | 53.7 | 46.3 | 48.4 | 40.7 | **69.4** | 42.9 | 14.2 |
| Qwen2.5-Coder-7B | 7B | 61.6 | 53.0 | 76.9 | 62.9 | 68.8 | **45.8** | **16.2** |
| Crystal-7B | 7B | 22.6 | 20.7 | 38.6 | 31.7 | 31.0 | 10.8 | 4.1 |
| StarCoder2-7B | 7B | 35.4 | 29.9 | 54.4 | 45.6 | 55.2 | 27.7 | 8.8 |
| StarCoder2-15B | 15B | 46.3 | 37.8 | 66.2 | 53.1 | 15.2 | 38.4 | 12.2 |
| OpenCoder-8B-Base | 8B | **66.5** | **63.4** | **79.9** | **70.4** | 60.6 | 40.5 | 9.5 |

Table 7: Performance of various base models on HumanEval, MBPP, and the "complete" task of BigCodeBench. Models trained on reproducible datasets are marked with **green**.

| Model | Size | HumanEval | | MBPP | | BigCodeBench | | LiveCodeBench |
|---|---|---|---|---|---|---|---|---|
| | | *HE* | *HE+* | *MBPP* | *MBPP+* | *Full* | *Hard* | *Avg* |
| **1B+ Models** | | | | | | | | |
| DS-coder-1.3B-Instruct | 1.3B | 65.2 | 61.6 | 61.6 | 52.6 | 22.8 | 3.4 | 9.3 |
| Qwen2.5-Coder-1.5B-Instruct | 1.5B | 70.7 | 66.5 | 69.2 | 59.4 | 32.5 | 6.8 | **15.7** |
| Yi-Coder-1.5B-Chat | 1.5B | 67.7 | 63.4 | 68.0 | 59.0 | 24.0 | 6.8 | 11.6 |
| OpenCoder-1.5B-Instruct | 1.5B | **72.5** | **67.7** | **72.7** | **61.9** | **34.6** | **11.5** | 12.8 |
| **6B+ Models** | | | | | | | | |
| DS-Coder-V2-Lite-Instruct | 16B | 81.1 | 75.0 | 82.3 | 68.8 | 36.8 | 16.2 | 24.3 |
| CodeLlama-7B-Instruct | 7B | 45.7 | 39.6 | 39.9 | 33.6 | 21.9 | 3.4 | 2.8 |
| CodeGemma-7B-It | 7B | 59.8 | 47.0 | 69.8 | 59.0 | 32.3 | 7.4 | 14.7 |
| DS-Coder-6.7B-Instruct | 6.7B | 78.6 | 70.7 | 75.1 | 66.1 | 35.5 | 10.1 | 20.5 |
| Yi-Coder-9B-Chat | 9B | 82.3 | 72.6 | 81.5 | 69.3 | 38.1 | 11.5 | 23.4 |
| CodeQwen1.5-7B-Chat | 7B | 86.0 | 79.3 | 83.3 | 71.4 | 39.6 | **18.9** | 20.1 |
| Qwen2.5-Coder-7B-Instruct | 7B | **88.4** | **84.1** | **83.5** | **71.7** | **41.0** | 18.2 | **37.6** |
| CrystalChat-7B | 7B | 34.1 | 31.7 | 39.1 | 32.7 | 26.7 | 2.3 | 6.1 |
| StarCoder2-15B-Instruct-v0.1 | 15B | 72.6 | 63.4 | 75.2 | 61.2 | 37.6 | 12.2 | 20.4 |
| OpenCoder-8B-Instruct | 8B | 83.5 | 78.7 | 79.1 | 69.0 | 42.9 | 16.9 | 23.2 |

Table 8: Performance of various chat models on HumanEval, MBPP, the "instruct" task of BigCodeBench and LiveCodeBench. Models trained on reproducible datasets are marked with **green**.

**GitHub star filtering limits diversity, reducing effectiveness** Github stars deteriorate performance (Allal et al., 2023). We further validate this conclusion and provide analysis from a visualization perspective. Specifically, we train two 1.5B LLMs, where one is trained original data and another is trained by data filtered by stars (stars>=5). As shown in Figure 5(c), star filter leads to decreased performance. We attribute this performance decline to the star filter's potential reduction of data diversity. As dedicated in Figure 5(d) and Figure 5(e), data applied star filter

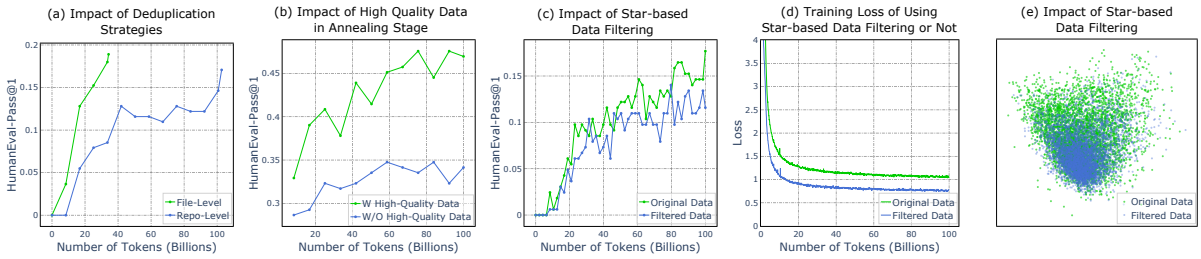


Figure 5: (a-c) shows 1.5B Code LLM performance across applying different ablation settings. (d) shows the comparison of training loss using star-based data filtering or not. (e) shows the distribution of data processed star-based data filtering or not.

| Dedup Level | Token(ratio) | HE | MBPP |
|---|---|---|---|
| File-Level | 32.7 (2.4%) | 18.9 | 19.4 |
| Repo-Level | 99.5 (7.3%) | 17.0 | 13.6 |

Table 9: Token counts and benchmark results using different deduplication strategies on RefineCode Python(1364B). HE/MBPP are obtained by training a 1.5B model for one epoch on all deduplicated tokens.

| | HE(+) | MBPP(+) | BCB | Arena |
|---|---|---|---|---|
| S1 | 52.4(48.1) | 68.7(57.4) | 22.1 | 5.3 |
| S2 | 69.1(64.0) | 69.5(60.3) | 32.6 | 5.8 |
| S1+2 | **72.5(67.7)** | **72.7(61.9)** | **34.6** | **6.9** |
| Mix | 55.5(51.2) | 52.0(58.7) | 23.9 | 3.8 |

Table 10: Performance of different training strategies across benchmarks. Mix Training refers to the process of combining and shuffling the data from Stage 1 and Stage 2 for joint training.

reflects a lower training loss and a more concentrated distribution, indicating that star filter significantly compromises data diversity. Upon closer examination of the filtered data, we find that it still contains a considerable amount of well-structured, algorithmically rich code. Therefore, we argue that using stars as a filtering criterion is not an optimal choice.

**Two-Stage SFT Strategy: First Broaden Knowledge, Then Sharpen Skills** Stage1 data exhibits significant diversity, while stage2 data shows higher quality. We believe this two-stage strategy enables the acquisition of broad capabilities in Stage 1, followed by targeted enhancement of code-related tasks in Stage 2. Besides, separating high-quality data can also prevent the gradient dilution problem that occurs when mixing datasets, maximizing the utilization of high-quality data. As shown in table 10, two-stage sft strategy can bring consistent improvement in both public benchmarks and real-word scenarios.

## 7 Related Work

**Open Large Language Models.** Recently, numerous open-sourced LLMs, such as LLaMA (Touvron et al., 2023), Mistral (Jiang et al., 2023), and Qwen (Bai et al., 2023), have empowered the research community, fostering innovation. Open datasets like RedPajama (Computer, 2023) and SlimPajama (Soboleva et al., 2023), MapNeo (Zhang et al., 2024a) alongside chat datasets such as WildChat (Zhao et al., 2024), further accelerate LLM advancements. Notably, fully open LLMs like OLMo (Groeneveld et al., 2024), OLMoE (Muennighoff et al., 2024), and LLM360 (Liu et al., 2023) provide comprehensive reproduction details, including data pipelines and checkpoints. In the realm of code LLMs, StarCoder (Allal et al., 2023) and StarCoderV2 (Lozhkov et al., 2024a) share high-quality pretraining corpora.

## 8 Conclusion

In this paper, we present OpenCoder, an open LLM specialized in code intelligence that achieves top-tier performance. To advance research transparency and reproducibility, we release our complete training materials, including: the complete data processing pipeline, the reproducible pretraining dataset, the synthetic dataset, the open code SFT dataset, rigorous experimental ablation results, detailed training protocols and intermediate checkpoints. The performance of OpenCoder is on par with leading proprietary models, and it surpasses most previous open-source models at the 1B+ and 6B+ parameter scale. We hope the release of OpenCoder can democratize access to all aspects of a top-tier code LLM, serving as both a powerful model and an open foundation to accelerate research and enable reproducible advancements in code AI.

## 9 Limitations

Although OpenCoder has explored the entire workflow for building a code LLM, our project still has several limitations. First, during the pretraining data filtering process, we focused exclusively on developing rule-based filtering rules but did not explore model-based filtering approaches. In addition, due to resource constraints, we only utilized raw code and code-related text data in the training process, without considering the potential promotion of general natural language data on the performance of code LLMs. Finally, in the post-training phase, we performed only supervised fine-tuning (SFT) on the base model, without considering reinforcement learning from human feedback (RLHF) for better alignment with human preferences.

## References


Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. 2023. Santacoder: don't reach for the stars! arXiv preprint arXiv:2301.03988.

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, abs/2108.07732.

BAAI. 2024. Hqcode dataset. Accessed: 2024-02-16.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming 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, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. 2023. Qwen technical report. arXiv preprint arXiv:2309.16609.

Andrei Z. Broder. 1997. On the resemblance and containment of documents. In Compression and Complexity of SEQUENCES 1997, Positano, Amalfitan Coast, Salerno, Italy, June 11-13, 1997, Proceedings, pages 21–29. IEEE.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. 2022. Multipl-e: A scalable and extensible approach to benchmarking neural code generation. arXiv preprint arXiv:2208.08227.

Linzheng Chai, Shukai Liu, Jian Yang, Yuwei Yin, Ke Jin, Jiaheng Liu, Tao Sun, Ge Zhang, Changyu Ren, Hongcheng Guo, et al. 2024. Mceval: Massively multilingual code evaluation. arXiv preprint arXiv:2406.07436.

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.

Cognitive Computations. 2023. Code-290k sharegpt vicuna dataset. https://huggingface.co/datasets/cognitivecomputations/Code-290k-ShareGPT-Vicuna.

Together Computer. 2023. Redpajama: an open dataset for training large language models.

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.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155.

Dirk Groeneveld, Iz Beltagy, Evan Walsh, Akshita Bhagia, Rodney Kinney, Oyvind Tafjord, Ananya Jha, Hamish Ivison, Ian Magnusson, Yizhong Wang, Shane Arora, David Atkinson, Russell Authur, Khyathi Chandu, Arman Cohan, Jennifer Dumas, Yanai Elazar, Yuling Gu, Jack Hessel, Tushar Khot, William Merrill, Jacob Morrison, Niklas Muennighoff, Aakanksha Naik, Crystal Nam, Matthew Peters, Valentina Pyatkin, Abhilasha Ravichander, Dustin Schwenk, Saurabh Shah, William Smith, Emma Strubell, Nishant Subramani, Mitchell Wortsman, Pradeep Dasigi, Nathan Lambert, Kyle Richardson, Luke Zettlemoyer, Jesse Dodge, Kyle Lo, Luca Soldaini, Noah Smith, and Hannaneh Hajishirzi. 2024. OLMo: Accelerating the science of language models. In Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 15789–15809, Bangkok, Thailand. Association for Computational Linguistics.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. 2023. Textbooks are all you need. arXiv preprint arXiv:2306.11644.

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.

Shengding Hu, Yuge Tu, Xu Han, Chaoqun He, Ganqu Cui, Xiang Long, Zhi Zheng, Yewei Fang, Yuxiang Huang, Weilin Zhao, et al. 2024. Minicpm: Unveiling the potential of small language models with scalable training strategies. arXiv preprint arXiv:2404.06395.

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.

INF-Team. 2024. Inf's open-source large language models.

Albert Qiaochu Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de Las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, L'elio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7b. ArXiv.

Katherine Lee, Daphne Ippolito, Andrew Nystrom, Chiyuan Zhang, Douglas Eck, Chris Callison-Burch, and Nicholas Carlini. 2021. Deduplicating training data makes language models better. arXiv preprint arXiv:2107.06499.

Jure Leskovec, Anand Rajaraman, and Jeffrey D. Ullman. 2014. Mining of Massive Datasets, 2nd Ed. Cambridge University Press.

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.

GitHub Linguist. 2024. languages.yml. Accessed: 2024-11-07.

Jiaheng Liu, Zhiqi Bai, Yuanxing Zhang, Chenchen Zhang, Yu Zhang, Ge Zhang, Jiakai Wang, Haoran Que, Yukang Chen, Wenbo Su, et al. 2024a. E2-llm: Efficient and extreme length extension of large language models. arXiv preprint arXiv:2401.06951.

Jiaheng Liu, Chenchen Zhang, Jinyang Guo, Yuanxing Zhang, Haoran Que, Ken Deng, Zhiqi Bai, Jie Liu, Ge Zhang, Jiakai Wang, et al. 2024b. Ddk: Distilling domain knowledge for efficient large language models. arXiv preprint arXiv:2407.16154.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024c. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. Advances in Neural Information Processing Systems, 36.

Shukai Liu, Linzheng Chai, Jian Yang, Jiajun Shi, He Zhu, Liran Wang, Ke Jin, Wei Zhang, Hualei Zhu, Shuyue Guo, et al. 2024d. Mdeval: Massively multilingual code debugging. arXiv preprint arXiv:2411.02310.

Zhengzhong Liu, Aurick Qiao, Willie Neiswanger, Hongyi Wang, Bowen Tan, Tianhua Tao, Junbo Li, Yuqi Wang, Suqi Sun, Omkar Pangarkar, Richard Fan, Yi Gu, Victor Miller, Yonghao Zhuang, Guowei He, Haonan Li, Fajri Koto, Liping Tang, Nikhil Ranjan, Zhiqiang Shen, Xuguang Ren, Roberto Iriondo, Cun Mu, Zhiting Hu, Mark Schulze, Preslav Nakov, Tim Baldwin, and Eric P. Xing. 2023. Llm360: Towards fully transparent open-source llms. Preprint, arXiv:2312.06550.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024a. Starcoder 2 and the stack v2: The next generation. arXiv preprint arXiv:2402.19173.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024b. Starcoder 2 and the stack v2: The next generation.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2024. Wizardcoder: Empowering code large language models with evol-instruct. In The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024. OpenReview.net.

Niklas Muennighoff, Luca Soldaini, Dirk Groeneveld, Kyle Lo, Jacob Morrison, Sewon Min, Weijia Shi, Pete Walsh, Oyvind Tafjord, Nathan Lambert, Yuling Gu, Shane Arora, Akshita Bhagia, Dustin Schwenk, David Wadden, Alexander Wettig, Binyuan Hui, Tim Dettmers, Douwe Kiela, Ali Farhadi, Noah A. Smith, Pang Wei Koh, Amanpreet Singh, and Hannaneh Hajishirzi. 2024. Olmoe: Open mixture-of-experts language models. Preprint, arXiv:2409.02060.

Guilherme Penedo, Hynek Kydlíček, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro Von Werra, Thomas Wolf, et al. 2024. The fineweb datasets: Decanting the web for the finest text data at scale. arXiv preprint arXiv:2406.17557.

Jim Plotts and Megan Risdal. 2023. Meta kaggle code.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. 2024. Chatdev: Communicative agents for software development. In Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 15174–15186.

Haoran Que, Jiaheng Liu, Ge Zhang, Chenchen Zhang, Xingwei Qu, Yi Ma, Feiyu Duan, Zhiqi Bai, Jiakai Wang, Yuanxing Zhang, Xu Tan, Jie Fu, Wenbo

Su, Jiamang Wang, Lin Qu, and Bo Zheng. 2024. D-cpt law: Domain-specific continual pre-training scaling law for large language models. ArXiv, abs/2406.01375.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, Y. Wu Y.K. Li, and Daya Guo. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models.

Noam Shazeer. 2020. Glu variants improve transformer. arXiv preprint arXiv:2002.05202.

Yikang Shen, Zhen Guo, Tianle Cai, and Zengyi Qin. 2024. Jetmoe: Reaching llama2 performance with 0.1 m dollars. arXiv preprint arXiv:2404.07413.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-lm: Training multi-billion parameter language models using model parallelism. Preprint, arXiv:1909.08053.

Daria Soboleva, Faisal Al-Khateeb, Robert Myers, Jacob R Steeves, Joel Hestness, and Nolan Dey. 2023. SlimPajama: A 627B token cleaned and deduplicated version of RedPajama.

Demin Song, Honglin Guo, Yunhua Zhou, Shuhao Xing, Yudong Wang, Zifan Song, Wenwei Zhang, Qipeng Guo, Hang Yan, Xipeng Qiu, et al. 2024. Code needs comments: Enhancing code llms with comment augmentation. arXiv preprint arXiv:2402.13013.

Tianhua Tao, Junbo Li, Bowen Tan, Hongyi Wang, William Marshall, Bhargav M Kanakiya, Joel Hestness, Natalia Vassilieva, Zhiqiang Shen, Eric P Xing, et al. Crystal: Illuminating llm abilities on language and code. In First Conference on Language Modeling.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. arXiv preprint arXiv:2302.13971.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. 2024. Opendevin: An open platform for ai software developers as generalist agents. arXiv preprint arXiv:2407.16741.

Zekun Moore Wang, Zhongyuan Peng, Haoran Que, Jiaheng Liu, Wangchunshu Zhou, Yuhan Wu, Hongcheng Guo, Ruitong Gan, Zehao Ni, Man Zhang, Zhaoxiang Zhang, Wanli Ouyang, Ke Xu, Wenhu Chen, Jie Fu, and Junran Peng. 2023. Rolellm: Benchmarking, eliciting, and enhancing role-playing abilities of large language models. arXiv preprint arXiv: 2310.00746.

Tianwen Wei, Liang Zhao, Lichang Zhang, Bo Zhu, Lijie Wang, Haihua Yang, Biye Li, Cheng Cheng, Weiwei Lü, Rui Hu, et al. 2023a. Skywork: A more open bilingual foundation model. arXiv preprint arXiv:2310.19341.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023b. Magicoder: Source code is all you need. arXiv preprint arXiv:2312.02120.

Yanan Wu, Jie Liu, Xingyuan Bu, Jiaheng Liu, Zhanhui Zhou, Yuanxing Zhang, Chenchen Zhang, Zhiqi Bai, Haibin Chen, Tiezheng Ge, et al. 2024. Conceptmath: A bilingual concept-wise benchmark for measuring mathematical reasoning of large language models. arXiv preprint arXiv:2402.14660.

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

Yuxiang630. 2024. Hqcode dataset. Accessed: 2024-02-16.

Ge Zhang, Scott Qu, Jiaheng Liu, Chenchen Zhang, Chenghua Lin, Chou Leuang Yu, Danny Pan, Esther Cheng, Jie Liu, Qunshu Lin, et al. 2024a. Map-neo: Highly capable and transparent bilingual large language model series. arXiv preprint arXiv:2405.19327.

Yifan Zhang, Yifan Luo, Yang Yuan, and Andrew Chi-Chih Yao. 2024b. Automathtext: Autonomous data selection with language models for mathematical texts. arXiv preprint arXiv:2402.07625.

Wenting Zhao, Xiang Ren, Jack Hessel, Claire Cardie, Yejin Choi, and Yuntian Deng. 2024. Wildchat: 1m chatGPT interaction logs in the wild. In The Twelfth International Conference on Learning Representations.

Fan Zhou, Zengzhi Wang, Qian Liu, Junlong Li, and Pengfei Liu. 2024. Programming every example: Lifting pre-training data quality like experts at scale. arXiv preprint arXiv:2409.17115.

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

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

# A Raw Code Processing

In this section, we delve into the details of the data processing pipeline in RefineCode. Specifically, we focus on on the design and considerations of two crucial modules: deduplication and filtering, and talk about their orders.

## A.1 Processing Details

**Deduplication** Owing to the extremely high repetition of the source code in Github, we adopt an aggressive file-level deduplication strategy (see elaborate analysis in Appendix C). More specifically, we leverage both exact deduplication and fuzzy deduplication methods to eliminate documents containing identical or near-identical code content shown as follows:

*Exact Deduplication*: Due to the prevalence of forking and copy-pasting within the codebase, nearly 75% of files are completely duplicated. On account of this, differing from general deduplication process, Identity removal is applied towards code data at the first step in this module. We compute the SHA256 hash value for each document, where files with identical hash values are compared, and only the code files with the highest star count as well as the latest commit time are retained, in order to preserve the highest quality and most up-to-date files.

*Fuzzy Deduplication*: This step aim to deduplicate those near-identical files. We split the raw text into 5-gram pieces, and then calculate the 2048 MinHash functions (Broder, 1997). Additionally, we utilize LSH (Leskovec et al., 2014) by setting bands to 16 and rows to 128, to retain only those distinct files with the highest stars and latest commit time. This process removes 6% file volume.

**Filtering** Given the distinct nature of code compared to natural language, the criteria for high-quality code differ significantly from those for natural language. Furthermore, different programming languages also exhibit distinct properties. Based on this, we believe that designing a set of detailed heuristic filtering rules tailored specifically to the characteristics of pretraining data is important to enhance the model's capabilities. Drawing inspiration from the principles of high-quality code data proposed in Gunasekar et al. (2023), we consider the following guidelines when designing our filters: **1) Filter out files with poor self-containment; 2) Filter out files with poor or minimal logical structure; 3) Remove files that deviate significantly from standard formatting.**

Based on these guidelines and the characteristics of our dataset, our work presents the first heuristic filtering framework by considering the unique characteristics of different programming languages. Based on RedPajama (Computer, 2023), this framework extends and refines the existing rules from StarCoder (Li et al., 2023) to better align with the unique properties of code datasets, resulting in more precise and higher-quality data cleansing. We developed the following three categories of filtering rules:

1. **Natural Language Filtering Rules**: These rules filter data based on common properties for all text files, such as file size, number of lines, and other general metrics. Both text and code files share these filtering rules.

2. **General Code Filtering Rules**: These rules apply to all code files by filtering data based on general code characteristics, such as the number of variables, average function length, and other common features.

3. **Language-Specific Filtering Rules**: These rules are designed according to the unique characteristics of specific programming languages, such as the frequency of "pass" statements in Python or the use of "goto" statements in C. We have developed these rules for the following eight commonly used programming languages: Python, C, C++, C#, Java, JavaScript, Go, and HTML.

Heuristic rules involve extensive threshold setting. When defining these rules and determining thresholds, we consistently follow a guiding principle: to remove harmful data as much as possible, while ensuring the overall distribution of the dataset is not significantly affected. We outline our motivations for rule design in Appendix B.1, along with a detailed explanation of the tuning process for the corresponding thresholds. Besides, we show the details of several representative rules in Appendix B.2.

## A.2 Processing Order

Most LLM data processing pipelines adopt a strategy where filtering is applied first, followed by deduplication. In contrast, our approach prioritizes deduplication before filtering, which offers advantages from two perspectives:

- **Processing Cost and Efficiency**: As mentioned earlier, over 90% of files in raw code are exact duplicates. Performing deduplication upfront helps avoid the computational overhead costing by filtering redundant files in the subsequent filtering phase. Additionally, the filtering rules are subject to frequent revisions, which means that both the filtering phase and the stages following it would need to be repeated. By conducting deduplication before filtering, we can mitigate the extra computational demands that arise from these repeated adjustments.

- **Data Intuition**: The effectiveness of the filtering stage must be evaluated based on the distribution of the processed data. Therefore, when filtering is applied last, the resulting data distribution directly reflects the distribution used during training, allowing for more intuitive and rapid adjustments to the filtering rules. In contrast, if filtering is applied before deduplication, the final data distribution used for training will be altered by the deduplication process, making it difficult to adjust the filtering rules based solely on the post-filtered data distribution.

Given these considerations, we argue that performing deduplication before filtering is a more rational choice for code pretraining data.

# B Filtering Rules

## B.1 Design of Filtering Rules

Designing heuristic filtering rules is inherently challenging, often requiring iterative refinement and experimentation to ultimately develop an effective set of rules. Given this complexity, in addition to providing detailed explanations of our designed rules, we will also share the general insights and methodologies we have accumulated throughout the designing process. We believe that this section will offer valuable guidance for designing heuristic filtering rules applicable to any dataset, thereby significantly enhancing the efficiency of constructing an effective data cleaning pipeline.

Heuristic rules filter data based on specific characteristics of a file, which, for each file, are ultimately expressed as a score representing the file's attribute and a corresponding threshold set by the rule. During the rule design process, we found that understanding the distribution of scores and the impact of different threshold settings on data filtering is critical to creating effective rules. Therefore, based on the approach used in RedPajama (Computer, 2023), we decompose the heuristic filtering process into two steps: **quality signal computation** and **filtering execution**. The quality signal computation calculates the scores for all rules for each file, while the filtering execution module decides whether a file is retained based on its quality signal scores and the corresponding thresholds.

Additionally, we recommend placing the heuristic filtering process as late as possible in the overall data pipeline. Unlike other, more fixed stages of the data processing pipeline, this stage requires frequent adjustments based on the final quality of the data. Placing it later in the process allows for more precise control over the data and minimizes the need to repeat subsequent steps after this filtering module.

The specific steps for designing our heuristic filtering rules are as follows:

1. **Quality Signals Designing:** Based on the definition of low-quality data and the attributes of the dataset, we firstly design a series of quality signals that describe the attributes contributing to file quality.

2. **Coarse Threshold Tuning:** Referring to the definition of low-quality data and the distribution of quality signal scores, we roughly set filtering thresholds for all rules at once. We then apply the filters to obtain an initial version of the filtered dataset.

3. **Fine-grained Threshold Tuning:** For each rule, we focus on the data that was exclusively affected by that specific rule, meaning it did not trigger other filters. This part of the data is directly influenced by the current rule, so we can examine whether the retention or removal of this data under different threshold settings aligns with the intended purpose of the rule. If a rule is effective in improving data quality based on its target attribute, we select the optimal threshold; otherwise, the rule is discarded. After evaluating each rule, we apply the filters again to obtain a more refined filtered dataset.

4. **Data Quality Inspection:** We then assess whether the filtered dataset meets our expectations for the quality of pretraining data. In addition to traditional manual inspection, we

introduce a perplexity (PPL)-based method for data quality evaluation. Specifically, we randomly sample a set of data from the filtered dataset and use a high-performing LLM to compute the PPL on these samples. We then examine the top-N and bottom-N samples based on PPL. Generally, extremely low PPL suggests that the data is overly simplistic, containing limited valuable knowledge, while extremely high PPL indicates that the data may lack learnable patterns. Both of them are advisable to be filtered out. We closely inspect both sets of samples and, based on their characteristics, decide whether to add new rules or adjust existing thresholds. This process can be repeated until the dataset reaches the desired quality.

### B.2 Examples of Filtering Rules

We elaborate several representative examples about general code filtering rules in Table 11 and language-specific filtering rules in Table 12 and explain their rationale. It is essential to note that for general code filtering rules, the threshold values may be slightly adjusted depending on the programming language of the file. For specific threshold values, please refer to our implementation details of the data processing pipeline.

## C Analysis on Chunk-level Deduplication

During pretraining, data is first randomly concatenated and segmented into chunks of context length, followed by full-attention computation within each chunk. We further explored chunk-level deduplication. Specifically, the pretraining data was randomly concatenated and segmented into chunks of 4096 tokens, followed by MinHash and LSH deduplication on these chunks. Additionally, we applied chunk-level deduplication after file-level and repo-level deduplication.

From the results in table 13, We observe that chunk-level deduplication alone was even less effective than repo-level deduplication, and applying chunk-level deduplication after file-level removed only an additional 0.04B of data. This indicates that chunk-level deduplication is not an effective approach. We pre-trained three 1.5B models on the data retained under file-level, repo-level, and repo-level + chunk-level deduplication strategies. The benchmark results are shown in Figure 6. It is evident that file-level deduplication achieves the highest training efficiency, while repo-level + chunk-level deduplication outperforms repo-level alone. We attribute the superior performance of file-level deduplication to its higher degree of data removal. Overall, we conclude that file-level deduplication is the most suitable method for GitHub data.

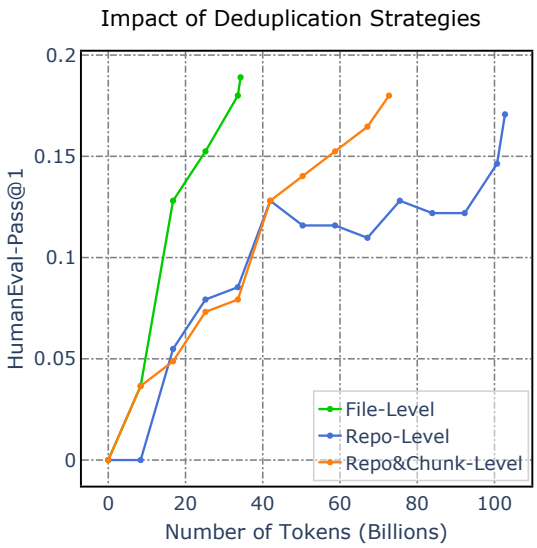


Figure 6: Comparison of Pass@1 performance on HumanEval & MBPP for different dedup strategies (File-Level, Repo-Level, and Repo-level + Chunk-Level) across RefineCode Python corpus.

## D Extra Data Processing

### D.1 Chinese Code-Like Domains Annotation

The manual annotation of the URLs of the website is presented as shown in the table 14. For future new CC datasets, we can sample pages in these domains as initial seed corpus.

### D.2 Code-Related Data from Github Text Files

Github Text files primarily consist of content written in natural languages, which includes abundant code-related knowledge. However, we observed that a substantial portion of the dataset is unrelated to code, which is detrimental to the model's ability to learn code-related knowledge. Therefore, we employed the following strategies to extract and retain the code-relevant portions before our filtering module. Firstly, following the strategy used in starcoder (Li et al., 2023), we retained the files with "requirement" in the lowercased filename, or if the filename without the extension is one of "readme", "notes", "todo", "description", "cmakelists", in order to ensure that only text files pertinent to coding contexts are preserved. This strategy recalled 3%

Table 11: Examples of general code filtering rules.

| Description | Explanation | Filtering Quota |
|---|---|---|
| The proportion of lines in strings with a word count exceeding. | Files with too many long strings indicate a lack of code logic. | score > 0.2 |
| The proportion of characters in words from strings with a character count exceeding 20. | String variables containing long sequences of characters are often indicative of meaningless content such as base64 data, Hash encoding, url, etc. | score > 0.4 |
| The proportion of hexadecimal characters. | Files with two many hexadecimal characters indicate a lack of code logic. | score > 0.4 |
| The proportion of lines like "you code here", "TODO" or "FIXME". | We found that these elements tend to be excessively repeated in the dataset, which increases the likelihood that the model, during code completion, will output placeholders like the ones mentioned above instead of generating actual code. | score > 0.01 |
| The proportion of lines containing an "assert" statement. | Files containing a large number of 'assert' statements are often test files, which tend to have relatively simple and repetitive code patterns. | score > 0.4 |

Table 12: Examples of python-specific filtering rules.

| Description | Explanation | Filtering Quota |
|---|---|---|
| The proportion of the number of python functions to the total number of lines. | A higher number of Python functions in a file may indicate that the functions are overly simple, with limited code logic, or have a bad code format. | score > 0.2 |
| Whether the file can be parsed into an python abstract syntax tree (AST). | Files that cannot be parsed into an AST contain syntax errors and should be filtered out. | score == False |
| The proportion of lines that are "import" statements. | A file with exceeding prportion of "import" statements indicates to have sparse code logic. | score > 0.3 |

volume of the whole text part. Additionally, we trained a fasttext model to recall code-related text files and recalled extra 7% file volume from the original text data.

### D.3 Jupyter Notebooks

Our Jupyter notebook data is sourced from GitHub and Meta Kaggle code (Plotts and Risdal, 2023). We converted this type of data into the *Jupyter-structured* format used in StarCoder (Li et al., 2023), which consists of a triplet of consecutive markdown, code, and code execution results. However, we discarded the *Jupyter-script* format mentioned in StarCoder. Because the code files generated from Jupyter notebook conversions tend to have poor overall code writing standards, and the content in *Jupyter-script* and *Jupyter-structured* formats is highly redundant, making it sufficient to retain only one format.

## E Programming Languages Categories

### E.1 Included Programming Languages

Included programming languages can be categoried into three classes: code, data and text. Among them, the "code" category represents files rich in code logic, while the "data" category primarily consists of files with structured data, and the "text" category refers to files dominated by natural language content. The threshold settings for the filtering rules vary slightly depending on the data type.

**Code(470 types):** 1C Enterprise, 4D, ABAP, ABAP CDS, AIDL, AL, AMPL, ANTLR, API

Table 13: Comparison of deduplication strategies on Python data. At the file-level, "Lines" refers to the number of lines in individual files; at the repo-level, it indicates the line count of aggregated strings; Note that for all deduplication strategies involving the Chunk level, "Lines" specifically refers to 4096-token chunks.

| Dedup Level | # Total Samples | # Retained Samples | # Retained Tokens |
|---|---|---|---|
| Chunk | 333,007,812 | 79,272,460 | 324.70 B |
| File | 485,817,123 | 30,488,834 | 32.74 B |
| File+Chunk | 333,007,812 | 7,993,164 | 32.70 B |
| Repo | 11,037,352 | 7,480,488 | 99.47 B |
| Repo+Chunk | 333,007,812 | 17,675,781 | 72.40 B |

Blueprint, APL, ASL, ASP.NET, ATS, ActionScript, Ada, Agda, Alloy, Alpine Abuild, AngelScript, Apex, Apollo Guidance Computer, AppleScript, Arc, AspectJ, Assembly, Astro, Asymptote, Augeas, AutoHotkey, AutoIt, Awk, BASIC, BQN, Ballerina, Batchfile, Beef, Befunge, Berry, Bikeshed, Bison, BitBake, Blade, BlitzBasic, BlitzMax, Bluespec, Boo, Boogie, Brainfuck, Brightscript, C, C#, C++, C2hs Haskell, CAP CDS, CLIPS, CMake, COBOL, CUE, Cadence, Cairo, CameLIGO, Cap'n Proto, Ceylon, Chapel, Charity, ChucK, Circom, Cirru, Clarion, Clarity, Classic ASP, Clean, Click, Clojure, Closure Templates, CodeQL, CoffeeScript, ColdFusion, ColdFusion CFC, Common Lisp, Common Workflow Language, Component Pascal, Coq, Crystal, Csound, Csound Document, Csound Score, Cuda, Curry, Cycript, Cypher, Cython, D, D2, DIGITAL Command Language, DM, Dafny, Dart, DataWeave, Dhall, Diff, Dockerfile, Dogescript, Dylan, E, ECL, EJS, EQ, Earthly, Edge, EdgeQL, Elixir, Elm, Elvish, Emacs Lisp, EmberScript, Erlang, F#, F*, FIRRTL, FLUX, Factor, Fancy, Fantom, Faust, Fennel, Filebench WML, Fluent, Forth, Fortran, Fortran Free Form, FreeBasic, Futhark, GAML, GAMS, GAP, GDB, GLSL, GSC, Game Maker Language, Genero 4gl, Genero per, Genshi, Gentoo Ebuild, Gentoo Eclass, Gherkin, Gleam, Glimmer JS, Glyph, Go, Golo, Gosu, Grace, Grammatical Framework, Groovy, Groovy Server Pages, HCL, HLSL, HTML, HTML+ECR, HTML+EEX, HTML+ERB, HTML+PHP, HTML+Razor, Hack, Haml, Handlebars, Harbour, Haskell, Haxe, HiveQL, HolyC, Hy, IDL, IGOR Pro, Idris, ImageJ Macro, Imba, Inform 7, Ink, Inno Setup, Io, Ioke, Isabelle, Isabelle ROOT, J, JCL, JFlex, JSONiq, Janet, Jasmin, Java, Java Server Pages, JavaScript, JetBrains MPS, Jinja, Jison, Jison Lex, Jolie, Jsonnet, Julia, Just, KRL, Kaitai Struct, KakouneScript, KerboScript, Kit, Kotlin, LFE, LLVM, LOLCODE, LSL, LabVIEW, Latte, Lean, Less, Lex, LigoLANG, LilyPond, Limbo, Liquid, Literate Agda, Literate CoffeeScript, Literate Haskell, LiveScript, Logos, Logtalk, LookML, Lua, Luau, M, M4, M4Sugar, MATLAB, MAXScript, MLIR, MQL4, MQL5, MTML, MUF, Macaulay2, Makefile, Mako, Marko, Mask, Mathematica, Mercury, Mermaid, Meson, Metal, MiniD, Mint, Mirah, Modelica, Modula-3, Module Management System, Mojo, Monkey, MoonScript, Motorola 68K Assembly, Move, Mustache, Myghty, NASL, NSIS, NWScript, Nearley, Nemerle, NetLinx, NetLogo, Nextflow, Nim, Nit, Nix, Nu, NumPy, Nunjucks, OCaml, Oberon, Objective-C++, Objective-J, Omgrofl, Opa, Opal, Open Policy Agent, OpenCL, OpenQASM, OpenSCAD, Ox, Oxygene, Oz, P4, PDDL, PEG.js, PHP, PLSQL, PLpgSQL, Pact, Pan, Papyrus, Parrot, Parrot Assembly, Parrot Internal Representation, Pascal, Pawn, Pep8, Perl, PigLatin, Pike, PogoScript, Polar, Pony, Portugol, PowerBuilder, PowerShell, Praat, Processing, Procfile, Prolog, Promela, Propeller Spin, Pug, Puppet, PureScript, Prover9, Pyret, Python, Q#, QML, QMake, Qt Script, Quake, R, RAML, REALbasic, REXX, RPGLE, RUNOFF, Racket, Ragel, Raku, Rascal, ReScript, Reason, ReasonLIGO, Rebol, Red, Redcode, RenderScript, Ring, Riot, RobotFramework, Roc, Rouge, Ruby, Rust, SAS, SMT, SQF, SQL, Sage, SaltStack, Sass, Scala, Scaml, Scenic, Scheme, Scilab, Self, Shell, ShellSession, Shen, Sieve, Singularity, Slash, Slim, Slint, SmPL, Smali, Smalltalk, Smarty, Smithy, Snakemake, SourcePawn, Squirrel, Stan, Standard ML, Starlark, Stata, Stylus, SugarSS, Svelte, Sway, Swift, SystemVerilog, TI Program, TL-Verilog, TLA, TSX, TXL, Talon, Tcl, Tcsh, Tea, Terraform Template, Thrift, Toit, Turing, Twig, TypeScript, Typst, Unified Parallel C, Uno, UnrealScript, UrWeb, V, VBA, VBScript, VCL, VHDL, Vala, Velocity Template Language, Verilog, Vim Script, Vim Snippet, Visual Basic .NET, Visual Basic 6.0, Volt, Vue, Vyper,

Table 14: We manually annotate code-like and math-like Chinese domains, utilizing the '%' symbol as a wildcard in our pattern matching. For example, the URL 'https://my.oschina.net/u/4/blog/11' is matched by the pattern '%my.oschina.net%blog%'.

| Domain | Prefix | Tag |
|---|---|---|
| cloud.tencent.com | %cloud.tencent.com/developer/article% | Code |
| cloud.tencent.com | %cloud.tencent.com/ask% | Code |
| cloud.tencent.com | %cloud.tencent.com/developer/information% | Code |
| cloud.tencent.com | %cloud.tencent.com/document% | Code |
| my.oschina.net | %my.oschina.net%blog% | Code |
| ask.csdn.net | %ask.csdn.net/questions% | Code |
| www.cnblogs.com | %www.cnblogs.com% | Code |
| forum.ubuntu.org.cn | %forum.ubuntu.org.cn% | Code |
| q.cnblogs.com | %q.cnblogs.com/q% | Code |
| segmentfault.com | %segmentfault.com/q% | Code |
| segmentfault.com | %segmentfault.com/a% | Code |
| woshipm.com | %woshipm.com/data-analysis% | Code |
| zgserver.com | %zgserver.com/server% | Code |
| zgserver.com | %zgserver.com/linux% | Code |
| zgserver.com | %zgserver.com/ubuntu% | Code |
| juejin.cn | %juejin.cn/post% | Code |
| jiqizhixin.com | %jiqizhixin.com/articles% | Code |
| help.aliyun.com | %help.aliyun.com/zh% | Code |
| jyeoo.com | %jyeoo.com% | Math |
| www.haihongyuan.com | %haihongyuan.com%shuxue% | Math |
| www.03964.com | %www.03964.com% | Math |
| www.nbhkdz.com | %www.nbhkdz.com% | Math |
| 9512.net | %9512.net% | Math |
| lanxicy.com | %lanxicy.com% | Math |
| bbs.emath.ac.cn | %bbs.emath.ac.cn% | Math |
| math.pro | %math.pro% | Math |
| mathschina.com | %mathschina.com% | Math |
| shuxue.chazidian.com | %shuxue.chazidian.com% | Math |
| shuxue.ht88.com | %shuxue.ht88.com% | Math |

WDL, WGSL, WebAssembly, WebIDL, Whiley, Witcher Script, Wollok, Wren, X10, XC, XProc, XQuery, XS, XSLT, Xojo, Xonsh, Xtend, YARA, YASnippet, Yacc, Yul, ZAP, ZIL, Zeek, ZenScript, Zephir, Zig, Zimpl, eC, fish, hoon, kvlang, mIRC Script, mcfunction, mupad, nesC, ooc, templ, wisp, xBase

**Data(115 types):** ABNF, ASN.1, Adobe Font Metrics, Altium Designer, Ant Build System, ApacheConf, Avro IDL, BibTeX, Browserslist, CIL, CODEOWNERS, CSON, CSS, Cabal Config, Caddyfile, CartoCSS, Cloud Firestore Security Rules, CoNLL-U, DNS Zone, Darcs Patch, Debian Package Control File, Dotenv, EBNF, Eagle, Easybuild, Ecere Projects, EditorConfig, Edje Data Collection, FIGlet Font, Formatted, GEDCOM, GN, Gemfile.lock, Gerber Image, Git Attributes, Git Config, Glyph Bitmap Distribution Format, Go Checksums, Go Module, Go Workspace, Godot Resource, Gradle, Gradle Kotlin DSL, GraphQL, Graphviz (DOT), HAProxy, HOCON, HTTP, HXML, INI, Ignore List, JAR Manifest, JSON, JSON with Comments, Jest Snapshot, Kusto, Lark, Linker Script, Maven POM, NEON, NL, NPM Config, Nginx, Ninja, ObjDump, Object Data Instance Notation, OpenStep Property List, OpenType Feature File, Option List, PlantUML, PostCSS, Prisma, Protocol Buffer, Protocol Buffer Text Format, Python traceback, RBS, RON, Readline Config, Record Jar, Redirect Rules, Regular Expression, SCSS, SELinux Policy, SPARQL, SSH

Config, STAR, STON, ShellCheck Config, Simple File Verification, Soong, Spline Font Database, TOML, TextMate Properties, Turtle, Type Language, Valve Data Format, Wavefront Material, Web Ontology Language, WebAssembly Interface Type, Wget Config, Windows Registry Entries, X BitMap, X Font Directory Index, XCompose, XML, XML Property List, XPages, YAML, YANG, cURL Config, crontab, desktop, dircolors, edn, nanorc

**Text(22 types):** AsciiDoc, Creole, Gemini, Gettext Catalog, MDX, Markdown, Muse, Org, Pod, Pod 6, RDoc, RMarkdown, Rich Text Format, Roff, SRecode Template, Sweave, TeX, Texinfo, Text, Textile, Wikitext, reStructuredText

### E.2 Excluded Programming Languages

2-Dimensional Array, AGS Script, Adblock Filter List, Bicep, COLLADA, CSV, Checksums, DirectX 3D File, E-mail, G-code, Git Revision List, Gnuplot, IRC log, KiCad Layout, KiCad Legacy Layout, KiCad Schematic, Lasso, Linux Kernel Module, Max, Microsoft Developer Studio Project, Microsoft Visual Studio Solution, POV-Ray SDL, Pic, Pickle, PostScript, Public Key, Pure Data, PureBasic, Raw token data, Roff Manpage, STL, SVG, SubRip Text, TSV, Unity3D Asset, Wavefront Object, WebVTT, X PixMap, robots.txt

## F Raw Code Data Composition

Figure 15 shows the composition of raw code data for top 85 programming languages in the **RefineCode** dataset, both after deduplication and filtering process, and Figure 7 unveil the training data compsition trending without data sampling. It can be observed that, after filtering, the proportion of data for different programming languages has shifted significantly, with a notable increase in the representation of commonly used programming languages.

## G Benchmark

### G.1 Benchmark for Base Models

**HumanEval & MBPP** We selected two widely used code completion benchmarks to evaluate OpenCoder, HumanEval (Chen et al., 2021)and MBPP (Austin et al., 2021). To further enhance the accuracy of the evaluation, EvalPlus (Liu et al., 2024c) extends HumanEval and MBPP to HumanEval Plus and MBPP Plus by adding unique and challenging test cases and correcting for inaccurate ground truth solutions. These results can be used to indicate the model's ability to understand and apply basic Python data structures and knowledge of algorithms. For HumanEval, we report the 0-shot results. For MBPP, we report 3-shots' results on 500 questions in the test split from MBPP (Austin et al., 2021), while the base and the plus results following EvalPlus (Liu et al., 2024c) report results on 378 questions in the sanitized part. Therefore, these results are not comparable and evaluated based on different data splits.

**BigCodeBench** BigCodeBench (Zhuo et al., 2024) is a challenging benchmark for code completion, designed to assess the models' ability to handle complex instructions and make accurate function calls across diverse external libraries. In the Completion setup, models are provided with a function signature, related documentation to generate appropriate code, and a unit test for the completed function. Covering a range of practical programming tasks, it evaluates the ability of the models to handle real-world scenarios involving complex, task-specific libraries.

### G.2 Benchmark for Instruct Model

**LiveCodeBench** LiveCodeBench is a comprehensive, contamination-free benchmark assessing highly complex algorithmic tasks' reasoning and problem-solving abilities. The benchmark is continuously updated with new problems from platforms such as LeetCode, AtCoder, and CodeForces, ensuring the challenges remain current and diverse. LiveCodeBench provides a robust measure of a model's ability to handle sophisticated logical processes, which is essential in competitive programming contexts. The instruct models are evaluated on the 2305-2409 data split.

We follow the Qwencoder evaluation code [1] to systematically measure performance in MultiPL-E (Cassano et al., 2022), providing insight into the adaptability and precision of the generation of LLM codes in Table 16. In addition, we evaluate our model using two more comprehensive benchmarks: McEval (Chai et al., 2024) in Table 8, and MdEval (Liu et al., 2024d) in Table 9.

**MultiPL-E** MultiPL-E extends the HumanEval benchmark to evaluate the code generation capabilities of large language models across multiple

[1] https://github.com/QwenLM/Qwen2.5-Coder

languages. MultiPL-E translates tasks into languages such as C++, Java, PHP, TypeScript, C#, Bash, and JavaScript, providing a consistent basis for assessing how models apply their programming skills across different syntaxes and paradigms. We follow the evaluation code of Qwencoder[2] to systematically measure performance in each language, providing insights into the adaptability and code generation accuracy of LLMs in a multilingual context.

**McEval** The comprehensive multilingual code evaluation benchmark McEval (Chai et al., 2024) employed a detailed assessment of OpenCoder's programming capabilities across 40 languages. In contrast to MultiPL-E, this benchmark is not derived from HumanEval or MBPP. Figure 8 depicts the results of the multilingual generation task for OpenCoder-8B-Instruct, which comprises nearly 2,000 samples. The figure illustrates that the model exhibits superior multilingual performance compared to other open-source models of comparable size.

**MdEval** OpenCoder is also evaluated on the comprehensive multilingual code debugging benchmark MdEval (Liu et al., 2024d) across 18 languages. In contrast to McEval, this benchmark focuses on the assessment of code debugging, especially for language-specific bugs. Figure 9 shows the results of the multilingual automated program repair task for OpenCoder-8B-Instruct, which comprises nearly 1.2K samples, which demonstrates that OpenCoder can effectively find the bugs and fix them compared to other open-source models of comparable size.

## H Prompts For SFT Synthetic Data

Prompts for generating synthetic code SFT data are shown below.

[2] https://github.com/QwenLM/Qwen2.5-Coder

Table 15: Overview of the data composition of in **RefineCode**. The items in the table are sorted in descending order according to the file volume after filtering.

| Language | After deduplication | | | After filtering | | |
|---|---|---|---|---|---|---|
| | # Files | Vol(GB) | Ratio(%) | # Files | Vol(GB) | Ratio(%) |
| html | 141,081,897 | 3,175.4 | 8.56 | 45,100,466 | 582.4 | 18.08 |
| java | 215,177,833 | 706.8 | 1.90 | 124,751,295 | 474.3 | 14.72 |
| python | 109,725,362 | 493.3 | 1.33 | 58,640,346 | 271.1 | 8.41 |
| csharp | 88,825,202 | 364.2 | 0.98 | 57,910,485 | 232.4 | 7.21 |
| javascript | 190,670,421 | 1,925.0 | 5.19 | 69,579,517 | 226.9 | 7.04 |
| php | 84,378,361 | 374.4 | 1.01 | 60,089,397 | 222.7 | 6.91 |
| cpp | 51,362,503 | 375.2 | 1.01 | 38,037,406 | 176.9 | 5.49 |
| go | 35,649,865 | 301.1 | 0.81 | 26,723,829 | 153.7 | 4.77 |
| typescript | 40,211,985 | 287.4 | 0.77 | 20,621,755 | 140.4 | 4.35 |
| ruby | 15,735,042 | 244.5 | 0.66 | 8,285,561 | 122.7 | 3.81 |
| perl | 16,354,543 | 121.7 | 0.33 | 9,532,620 | 65.6 | 2.04 |
| rust | 10,605,421 | 63.6 | 0.17 | 6,086,150 | 39.9 | 1.24 |
| r | 6,132,978 | 92.5 | 0.25 | 4,803,109 | 34.7 | 1.08 |
| swift | 4,238,754 | 47.9 | 0.13 | 2,938,498 | 31.8 | 0.99 |
| kotlin | 4,493,548 | 56.4 | 0.15 | 3,123,156 | 29.8 | 0.94 |
| dart | 4,087,329 | 33.0 | 0.09 | 2,161,462 | 18.5 | 0.57 |
| java-pages | 6,174,654 | 31.0 | 0.08 | 4,145,336 | 15.4 | 0.48 |
| css | 39,822,744 | 241.5 | 0.65 | 15,771,061 | 15.3 | 0.47 |
| lua | 4,027,221 | 116.0 | 0.31 | 2,538,234 | 14.4 | 0.45 |
| xml | 61,171,289 | 1,934.2 | 5.21 | 3,173,128 | 12.8 | 0.40 |
| scala | 5,897,567 | 19.7 | 0.05 | 4,204,979 | 11.7 | 0.36 |
| shell | 12,054,632 | 23.0 | 0.06 | 6,043,070 | 11.2 | 0.35 |
| pascal | 1,306,130 | 27.8 | 0.07 | 960,497 | 9.5 | 0.29 |
| fortran | 2,274,663 | 39.7 | 0.10 | 1,218,491 | 8.6 | 0.27 |
| perl6 | 1,943,430 | 16.4 | 0.04 | 1,034,748 | 8.6 | 0.27 |
| rmarkdown | 1,317,760 | 14.0 | 0.04 | 827,951 | 7.9 | 0.25 |
| html+erb | 7,618,377 | 11.4 | 0.03 | 4,452,355 | 7.8 | 0.24 |
| smali | 3,457,531 | 37.9 | 0.10 | 1,408,274 | 7.4 | 0.23 |
| scss | 18,061,278 | 35.6 | 0.10 | 7,705,822 | 7.4 | 0.23 |
| gettext catalog | 1,100,044 | 51.3 | 0.14 | 442,385 | 6.3 | 0.19 |
| haskell | 1,746,444 | 24.0 | 0.06 | 1,218,491 | 6.8 | 0.27 |
| tcl | 253,345 | 4.2 | 0.01 | 136,171 | 1.0 | 0.03 |
| gradle | 2,431,985 | 2.9 | 0.01 | 724,609 | 1.0 | 0.03 |
| scheme | 357,909 | 4.7 | 0.01 | 201,170 | 1.0 | 0.03 |
| qml | 354,756 | 1.8 | 0.01 | 252,621 | 1.0 | 0.03 |
| mdx | 795,525 | 6.4 | 0.17 | 222,013 | 1.0 | 0.03 |
| classic asp | 220,344 | 2.8 | 0.08 | 141,236 | 0.9 | 0.03 |
| xbase | 192,780 | 2.5 | 0.07 | 80,396 | 0.9 | 0.03 |
| ini | 7,232,136 | 19.1 | 0.05 | 1,517,099 | 1.3 | 0.04 |
| objective-c++ | 197,416 | 2.4 | 0.01 | 149,223 | 1.3 | 0.04 |
| motorola68k | 1,066,095 | 26.5 | 0.07 | 220,218 | 1.2 | 0.04 |
| gap | 752,261 | 2.6 | 0.01 | 510,420 | 1.2 | 0.04 |

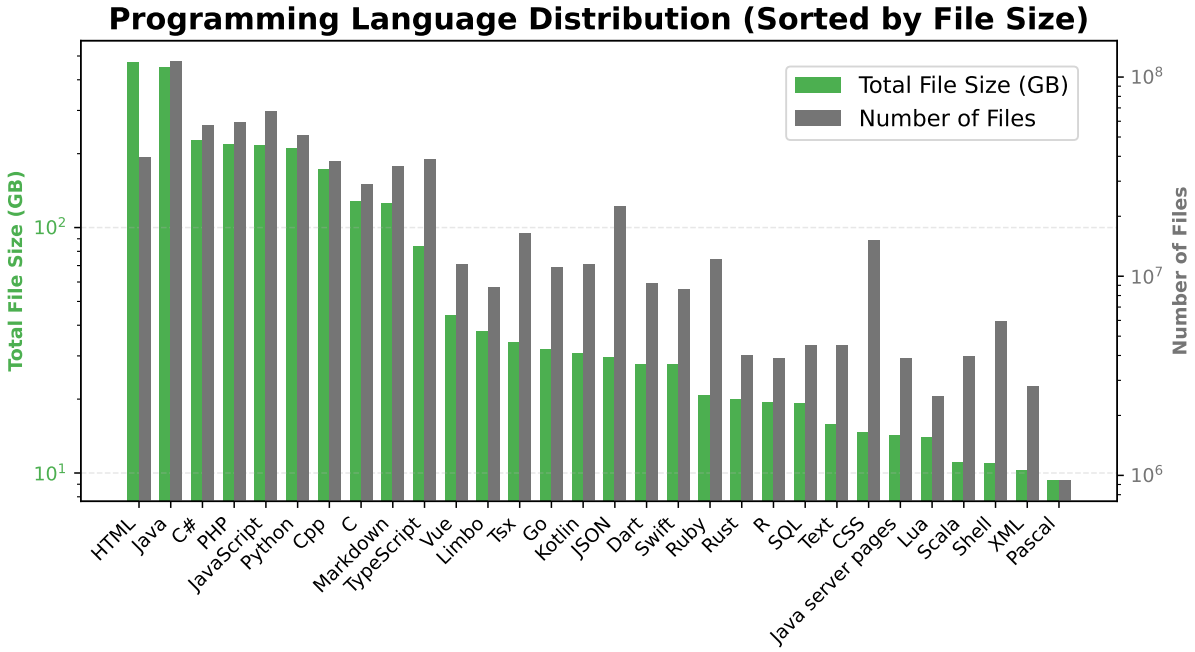


Figure 7: The distribution of top program languages in **RefineCode** (before data sampling).

| **Model** | **Size** | Python | Java | C++ | C# | TS | JS | PHP | Bash | **Average** |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | **1B+ Models** | | | | | | |
| DS-Coder-1.3B-Instruct | 1.3B | 65.2 | 51.9 | 45.3 | 55.1 | 59.7 | 52.2 | 45.3 | 12.7 | 48.4 |
| Yi-Coder-1.5B-Chat | 1.5B | 67.7 | 51.9 | 49.1 | 57.6 | 57.9 | 59.6 | 52.2 | 19.0 | 51.9 |
| Qwen2.5-Coder-1.5B-Instruct | 1.5B | 71.2 | 55.7 | **50.9** | **64.6** | 61.0 | **62.1** | **59.0** | 29.1 | 56.7 |
| OpenCoder-1.5B-Instruct | 1.5B | **72.5** | **64.6** | **50.9** | 61.4 | **63.5** | **62.1** | 55.3 | **29.7** | **57.5** |
| | | | | **6B+ Models** | | | | | | |
| DS-Coder-6.7B-Instruct | 6.7B | 78.6 | 68.4 | 63.4 | 72.8 | 67.2 | 72.7 | 68.9 | 36.7 | 66.1 |
| DS-Coder-V2-Lite-Instruct | 16B | 81.1 | **76.6** | **75.8** | 76.6 | 80.5 | 77.6 | 74.5 | 43.0 | 73.2 |
| CodeLlama-7B-Instruct | 7B | 45.7 | 32.2 | 28.6 | 32.9 | 39.0 | 43.5 | 31.7 | 10.1 | 33.0 |
| CodeGemma-7B-It | 7B | 59.8 | 48.1 | 46.6 | 51.9 | 54.7 | 54.0 | 46.6 | 10.1 | 46.5 |
| CodeQwen1.5-7B-Chat | 7B | 83.5 | 70.9 | 72.0 | 75.9 | 76.7 | 77.6 | 73.9 | 41.8 | 71.6 |
| Yi-Coder-9B-Chat | 9B | 85.4 | 76.0 | 67.7 | 76.6 | 72.3 | 78.9 | 72.1 | 45.6 | 71.8 |
| Qwen2.5-Coder-7B-Instruct | 7B | **87.8** | 76.5 | 75.6 | **80.3** | **81.8** | **83.2** | **78.3** | **48.7** | **76.5** |
| OpenCoder-8B-Instruct | 8B | 83.5 | 72.2 | 61.5 | 75.9 | 78.0 | 79.5 | 73.3 | 44.3 | 71.0 |

Table 16: Performance of various chat models on the MultiPL-E benchmark across different programming languages.

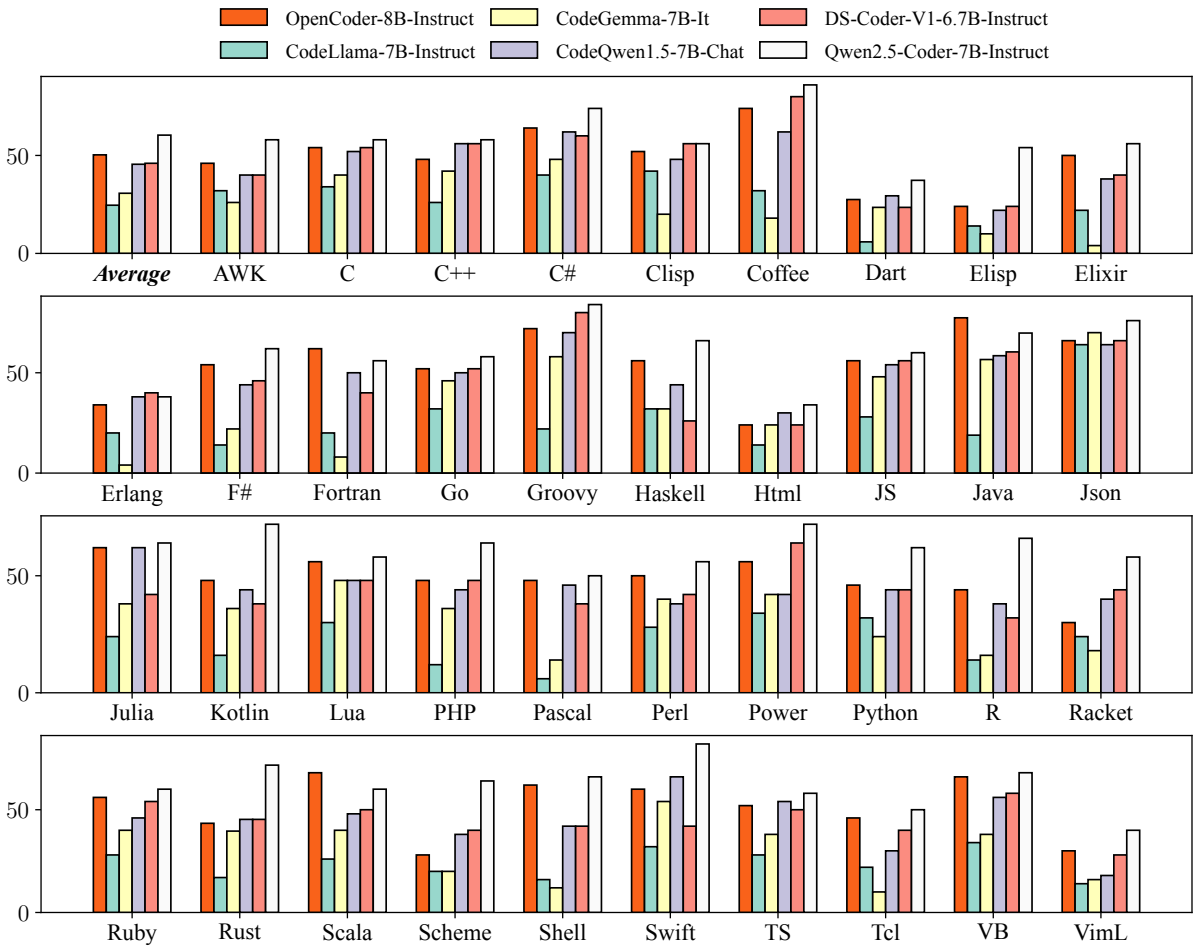


Figure 8: The McEval performance of OpenCoder-8B-Instruct in comparison to other open-source code models of comparable size.

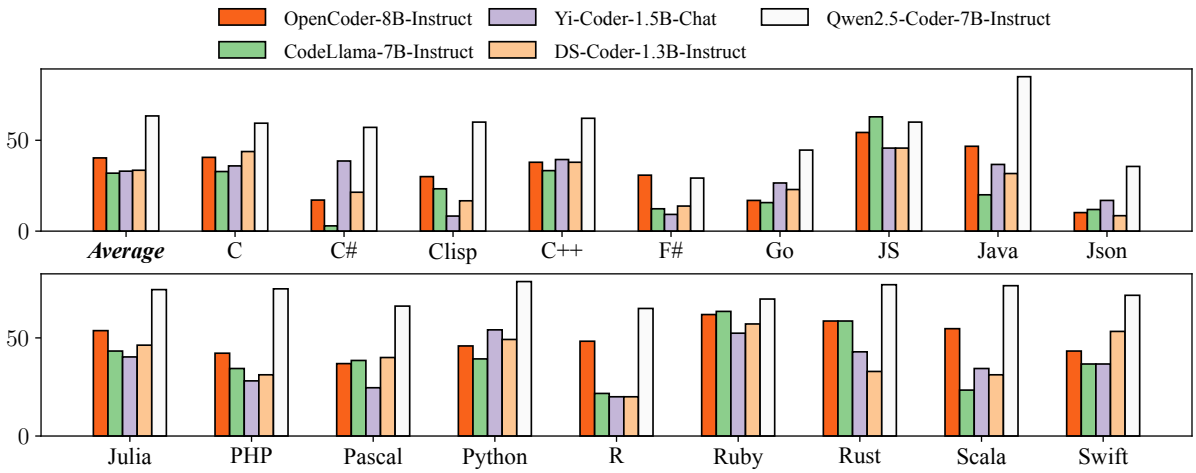


Figure 9: The MdEval performance of OpenCoder-8B-Instruct in comparison to other open-source code models of comparable size.

Prompt for Educational Instruction Synthesis

You are a teaching assistant helping to create a Python programming task from a given code snippet. You must provide the best response to the Python programming task, including reasoning thought, reference solutions, explanation of test cases, and test code.
**[Code Snippet]**
{Code}

Your response must have these parts:

**[Task]**
{Create an independent and detailed Python programming task}

**[Analysis]**
{Analyze the task and reason about the given task step by step}

**[Solution]**
{Write a high-quality reference solution in a self-contained script that solves the task}

**[Test]**
{Provide ten assert statements to check the correctness of your solution}

Prompt for Package-related Instruction Synthesis

You are exceptionally skilled at crafting high-educational level problems and offering precise solutions. Please gain inspiration from the following code snippet to create a high-quality programming problem, which is beneficial for learning the use of corresponding libraries. Present your output in two distinct sections: [Problem Description] and [Solution].

**[Code Snippet]**
{Code}

**[Library Api Requirements]**
{Api Requirements}

**[Library Api Doc]**
{Api Doc}

Guidelines for each section:
1. [Problem Description]: This should be **completely self-contained**, providing all the contextual information one needs to understand and solve the problem. Assume common programming knowledge, but ensure that any specific context, variables, or code snippets pertinent to this problem are explicitly included. This problem should be **educational for learning the provided Library api, and please explicitly request the use of the relevant package in the question. This question should only concern the writing of **one function**, and you need to be clear about the function name and role of this function.

2. [Solution]: Offer a comprehensive, **correct** solution that addresses the [Problem Description] you provided. This solution should follow the standard of corresponding Library Api doc. Please ensure that the Solution only involves answering the Problem, **without addressing the requirements I provided!** Please provide essential explanation abouth this solution, especially the use of requiremed Library Api.

Prompt for Large-scale Diverse Instruction Synthesis

You are an expert in designing high-quality programming questions based on the given text.

**[Guidelines]**
- You can draw inspiration from the given text to create the programming questions.
- The created question should be a self-contained question, which does not depend on any external context.
- The created response must contain the complete code snippet.

**[Given Text]**
{Given Text}

**[Created Question]**
{Created Question}