🌓 Seed-Coder: Let the Code Model Curate Data for Itself

ByteDance Seed

🌐 Homepage   ⭘ GitHub   🤗 Hugging Face

Abstract

Code data in large language model (LLM) pretraining is recognized crucial not only for code-related tasks but also for enhancing general intelligence of LLMs. Current open-source LLMs often heavily rely on human effort to produce their code pretraining data, such as employing hand-crafted filtering rules tailored to individual programming languages, or using human-annotated data to train quality filters. However, these approaches are inherently limited in scalability, prone to subjective biases, and costly to extend and maintain across diverse programming languages. To address these challenges, we introduce Seed-Coder, a series of open-source LLMs comprising base, instruct and reasoning models of 8B size, minimizing human involvement in data construction. Our code pretraining data is produced by a model-centric data pipeline, which predominantly leverages LLMs for scoring and filtering code data. The instruct model is further trained via supervised fine-tuning and preference optimization, and the reasoning model leverages Long-Chain-of-Thought (LongCoT) reinforcement learning to improve multi-step code reasoning. Seed-Coder achieves state-of-the-art results among open-source models of similar size and even surpasses some much larger models, demonstrating superior performance in code generation, code completion, code editing, code reasoning, and software engineering tasks.
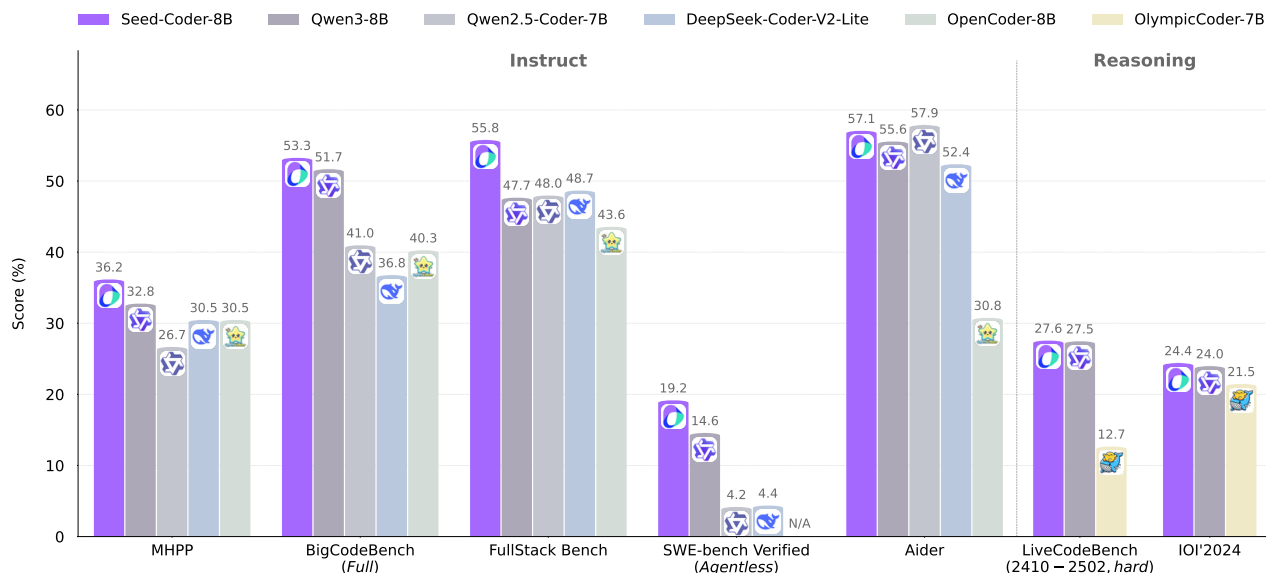
Figure 1. Benchmark performance of instruct and reasoning variants of Seed-Coder-8B.

Contents

1. Introduction

Large language models (LLMs) have achieved superior performance in a wide range of coding tasks, showing their potential to revolutionize the entire ecosystem of software development. Current state-of-the-art models, such as Claude 3.7 Sonnet [Anthropic, 2025] and OpenAI o3 [OpenAI, 2025], have demonstrated unprecedented capability in a wide range of code-related tasks, including code generation, code explanation, code debugging, code editing, and real-world software engineering tasks. These models can largely enhance developers' productivity and provide substantial support for the software industry, showing the potential to gradually automate the process of software development. However, these cutting-edge models remain proprietary and have disclosed very little about their training data.

State-of-the-art open-source LLMs, such as DeepSeek-R1 [DeepSeek-AI et al., 2025] and Qwen2.5-Coder [Hui et al., 2024], have invigorated the research community by offering self-deployable models that demonstrate competitive performance on coding tasks. While their technical reports provide valuable insights into both pretraining and post-training techniques, many offer only a high-level overview of their data processing methodologies. Among these ideas, one broad consensus for data quality control lies in that leveraging human expertise to curate and refine datasets is an intuitive and effective approach to enhancing the quality of code pretraining data. For example, DeepSeek-Coder [Guo et al., 2024] and DeepSeek-Coder-V2 [DeepSeek-AI et al., 2024a] apply a set of filtering rules as the initial data processing step, following the filtering rules used in StarCoder [Li et al., 2023]. Qwen2.5-Coder [Hui et al., 2024] also utilizes a series of rule-based filtering methods similar to DeepSeek-Coder. Moreover, OpenCoder [Huang et al., 2025] incorporates over 130 hand-crafted filtering rules with customized weights in their pretraining data pipeline.

However, hand-crafted rules are prone to conflicts among themselves and may incur high costs to extend and maintain across diverse programming languages. Existing human-centric data filtering approaches that heavily rely on hand-crafted rules often introduce subjective biases and inherently lack scalability. We believe that these limitations align with insights from the widely cited essay "The Bitter Lesson" [Sutton, 2019]: AI researchers often favor human-centric methods due to their short-term advantages, yet these approaches inevitably plateau and even inhibit long-term progress; the breakthrough progress eventually arrives by the opposing approach via scaling computation and data. In the context of code pretraining data, the human-centric approaches appear particularly favorable, since many AI researchers are skilled programmers themselves and feel confident in evaluating code quality. Nevertheless, human-centric methods would ultimately prove restrictive and tend to impede advancements of code LLMs in the long run.

In this report, we present a model-centric data pipeline that minimizes human involvement in constructing code pretraining data. Our data pipeline predominantly leverages LLMs, instead of hand-crafted rules, for scoring and filtering code data. Compared to human-centric approaches, the use of LLMs is particularly advantageous, as they effectively capture nuanced standards of code quality that are difficult to quantify explicitly, and simultaneously provide scalable, self-contained evaluations capable of processing billions of samples consistently. We applied these LLM filters during the preprocessing of our data – including GitHub code, GitHub commits, and code-related web data – and curated a code pretraining corpus comprising a total of 6 trillion tokens.

Building on this foundation, we introduce Seed-Coder, a family of state-of-the-art open-source code LLMs at the 8B scale, including a base model, an instruct model, and a reasoning model.
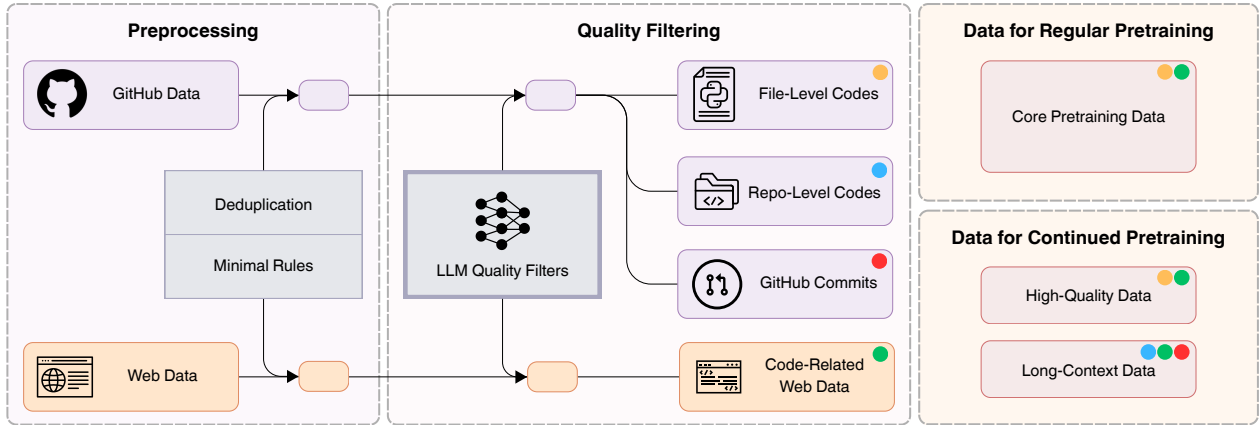
Figure 2. Processing pipeline for pretraining data. We collected data from GitHub and web archives. The raw data were processed into four categories: file-level codes (yellow), repository-level codes (blue), GitHub commits (red) and code-related web data (green). For each phase in pretraining, we combined and reorganized the processed data from the four categories, indicated by colors on the top-right of the blocks.

Starting from the base model trained with the proposed pretraining pipeline, the instruct model is further fine-tuned on large-scale synthetic data generated and filtered by LLMs, followed by direct preference optimization (DPO) to enhance the instruction-following capabilities. Meanwhile, the reasoning model improves multi-step reasoning in complex coding tasks by applying Long-Chain-of-Thought (LongCoT) reinforcement learning. Extensive evaluations on well-established benchmarks on a diverse set of coding tasks were conducted to showcase the advanced capabilities of the three variants of Seed-Coder. We release this lightweight yet powerful model family with the intention of providing research insights while maintaining a manageable scale to facilitate further exploration within the open-source community.

2. Pretraining

2.1. Data Pipeline

Producing data for LLM pretraining typically requires complex data pipelines and intricate data lineage. To ensure efficiency and reduce computation and storage costs, we propose a parallel design, which decouples the sequential dependency of various filters so as to streamline our data pipeline. As illustrated in Figure 2, our end-to-end data pipeline takes as input the raw code data crawled from GitHub and web archives, and outputs the final pretraining data after a few steps of data processing. All the preprocessing and filtering modules were disentangled so that each can run individually to avoid re-running the entire pipeline for better support of incremental data expansion and flexible pipeline manipulation.

Our data pipeline starts with the standard exact- and near-deduplication to get a holistic set of code-related files without duplication. Then we applied basic filters with minimal rules such as language inferring to get rid of irrelevant and non-code data. On top of the preprocessed data, we developed advanced quality filters powered by LLMs, which capture general standards and exhibit

strong generalizability to address the wide variety across the large-scale data. Our filtered data falls into four categories:

- **File-level codes:** Individual code files from GitHub Data.

- **Repository-level codes:** Code files structured based on the repositories.

- **Commits data:** Snapshots of GitHub Commits, including commit messages, repository metadata, all relevant files, and code patches.

- **Code-related web data:** Documents from web archives that contains code blocks or highly code-related.

Based on the four categories above, we designed our pretraining recipes mainly comprising two phases: In the regular pretraining phase, we used file-level codes and code-related web data to build the fundamental capabilities of our model; in the continued pretraining phase, we expanded to all four categories to enhance performance and alignment, while also stimulating the model's ability to understand long-context data.

In the next section, we detail the pipeline over individual data ingredients. Section 2.2.1 introduces the pipeline for GitHub data, with an emphasis on the general design and implementation of our LLM quality filters. Sections 2.2.2 and 2.2.3 present the processing of GitHub commits and web archives, respectively. Sections 2.2.4 and 2.2.5 describe the construction of the continued pretraining data. Finally, we conclude the pretraining section with discussions on the Fill-in-the-Middle (FIM) format applied to both phases.

2.2. Data Ingredients

2.2.1. GitHub Data

GitHub is widely regarded as the most abundant and valuable data source for training LLM coders. In our work, it serves as the primary component of our pretraining corpus. The raw data was collected in the form of repositories, and we adopted the two-stage processing pipeline illustrated in Figure 2 to extract a high-quality subset.

Preprocessing. We commenced by implementing deduplication at both the repository and file levels. For each level, we performed exact-deduplication using SHA256 hashes of contents and near-deduplication via the MinHash algorithm [Kiveris et al., 2014, Lee et al., 2022]. This two-tier strategy yielded two variants of the code corpus: the file-level variant offered flexibility for training with short context windows, while the repository-level variant preserved the project structure, enabling more coherent long-context learning. Following the deduplication, we checked the remaining files via syntax parsers such as Tree-sitter [Brunsfeld, 2018] and discarded those with syntax errors. Overall, the preprocessing stage reduced the raw data volume by approximately 98%, resulting in a manageable dataset for downstream quality filtering.

Quality Filtering. Rule-based filters are widely used to enhance the quality of pretraining datasets. The rules defined by coding experts provide strong controllability and interpretability in high-quality document selection. However, these filters are inherently limited to rules that can be precisely defined, making it difficult to incorporate empirical heuristics. They also lack flexibility in adjusting filtering strength. More fundamentally, rule creation relies on expert consensus, a stringent

```python
def main(temp):
    """
    Display the message according to the following conditions
    Temp < 0: "Freezing"
    Temp 1-10: "Very Cold"
    Temp 11-20: "Cold"
    Temp 21-30: "Normal"
    Temp 31-40: "Hot"
    Temp > 40: "Very Hot"
    Args:
        temp: integer
    Returns:
        string: the message to print
    """
    if temp < 0 and temp > 1:
        print("Freezing")
    if temp > 0 and temp < 11:
        print("Very Cold")
    if temp > 10 and temp < 21:
        print("Cold")
    if temp > 20 and temp < 31:
        print("Normal")
    if temp > 30 and temp < 41:
        print("Hot")
    if temp > 40:
        print("Very Hot")
    return
```

Figure 3. Sample Python script with decent structure but logical errors.

requirement that may not always be met. A notable example in code datasets is class templates with extensive comments. Some may argue that detailed comments help the model understand the semantic meanings and usage of class functions and attributes. Others, however, may see them as diluting code-related content, potentially hindering coding capabilities by overemphasizing plain text rather than actual code lines. Similarly, a short document consisting only of API calls may be dismissed for its brevity and lack of logical statements, yet it can also be valued as a fundamental building block in software development.

Beyond the challenges of constructing rule-based filters, heuristics derived from human expertise can also be unreliable, as fully assessing a code document requires a comprehensive grasp of its entire context, which is an exhaustive and inherently unscalable task. Figure 3 illustrates this issue with a function that appears heuristically sound at first glance, featuring well-commented usage and a structured pipeline. However, it harbors subtle logical errors that are easily overlooked. Figure 4 presents an even more extreme case, where human experts struggle to decipher the script's meaning unless viewed from a significant distance.

To overcome the above shortness of human-oriented filter designs, we propose a file-level scoring model to filter out low-quality code files from GitHub data in one shot, freeing us from fabricating the complicated standards one by one. The pipeline for constructing and using the scorer as a quality filter is shown in Figure 5. To construct the specific training set for this filter, we randomly sampled $222,066$ code files from the most commonly used programming languages and queried DeepSeek-V2-Chat [DeepSeek-AI et al., 2024a] (conceptually referred to as "the oracle") to assess four key aspects: readability, modularity, clarity, and reusability.

- **Readability:** High-quality code should include a reasonable amount of comments, follow consistent naming conventions, and adhere to common formatting and structural practices.

- **Modularity:** High-quality code should be well-structured, avoiding overly complex or lengthy functions by leveraging modularization. Each module or component should serve a distinct, coherent purpose with a clear separation of logic and functionality.

7

```
t3 = [
    [0, 0, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7, 0, 0],
    [0, 7, 7, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7, 7, 7, 0],
    [0, 7, 7, 7, 0, 0, 0, 0, 0, 0, 0, 0, 7, 7, 7, 0],
    [0, 0, 7, 7, 7, 0, 0, 0, 0, 0, 0, 7, 7, 7, 0, 0],
    [0, 0, 0, 7, 7, 7, 0, 0, 0, 0, 0, 7, 7, 0, 0, 0],
    [0, 0, 0, 7, 7, 7, 0, 0, 0, 0, 7, 7, 0, 0, 0, 0],
    [0, 0, 0, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 0, 0, 0],
    [0, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 0],
    [0, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 0],
    [0, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 0],
    [0, 7, 7, 7, 0, 7, 7, 7, 7, 7, 0, 7, 7, 7, 7, 0],
    [0, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 0],
    [0, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 0],
    [0, 7, 7, 7, 7, 7, 7, 0, 0, 7, 7, 7, 7, 7, 7, 0],
    [0, 0, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 0, 0],
    [0, 0, 0, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 0, 0, 0],
    [0, 0, 0, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 0, 0, 0],
    [0, 0, 0, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 0, 0, 0],
    [0, 0, 0, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 0, 0, 0],
    [0, 0, 0, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 0, 0, 0],
    [0, 0, 0, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 0, 0, 0],
    [0, 0, 0, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 0, 0, 0],
    [0, 0, 0, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 0, 0, 0],
    [0, 0, 0, 0, 7, 7, 7, 0, 0, 7, 7, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 7, 7, 0, 0, 7, 7, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 7, 7, 7, 0, 0, 7, 7, 7, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
]
```

Figure 4. Sample code snippet from a Python script for LED display controlling. The original code (left) and a visually intuitive format by replacing zeros and commas with space (right) are presented.

- **Clarity:** High-quality code should minimize redundancy, such as excessive function calls and large blocks of commented-out code or debugging print statements. It should clearly convey the intent behind each code block.

- **Reusability:** High-quality code should be free of syntax and logical errors, avoid excessive hard-coded data, and be designed for easy integration into other projects with complete and meaningful functionality.

These standards primarily apply to substantial code files. However, randomly sampled code files may also contain incidental content such as configurations, data, or documents. To mitigate this, we applied high-level constraints to exclude such cases. Additionally, we filtered out auto-generated code, which often contains repetitive blocks and may introduce rigidity during pretraining. The final prompt for evaluating the quality of GitHub code files can be found in Appendix A.2.1.

The oracle was required to give an overall score ranging from 0 to 10 that evaluates the quality of code files (higher indicates better quality), with detailed explanations to support the score. Subsequently, only the score itself was extracted from the oracle's response to serve as the ground-truth label for each file. To scale the scorer to the full pretraining GitHub dataset, we opted for a regression model rather than a binary classifier. This approach allows for finer-grained quality evaluation, avoiding the rigidity of rule-based pass-or-fail filtering. Prioritizing efficiency for large-scale inference, we rescaled the ground-truth scores to the range [0,1] and fine-tuned a pretrained Llama 2 model [Touvron et al., 2023] of size 1.3B with a regression head for one epoch as the quality scorer. To balance the quality and diversity of code pretraining data, we filtered out from the entire GitHub dataset the bottom ~10% files, aggregating to ~1T unique tokens based on this quality scoring method. This corpus supports 89 programming languages, forged into both the repository-level and file-level code data shown in Figure 2. The list of supported languages can be found in Appendix A.1. Figure 6 offers a glimpse into the effectiveness of LLM-based filters by presenting a snapshot of benchmark performance during pretraining. Further details on oracle selection, ground-truth generation, model architecture, and ablation studies can be found in Appendix A.2.
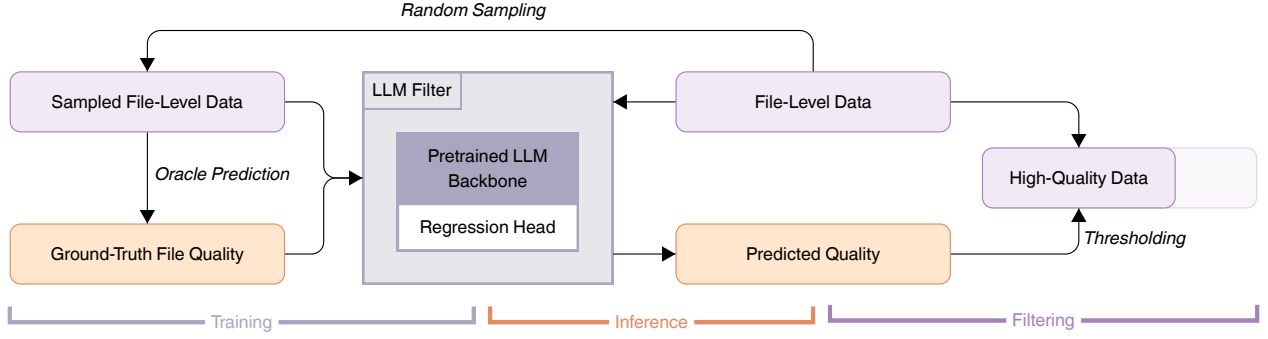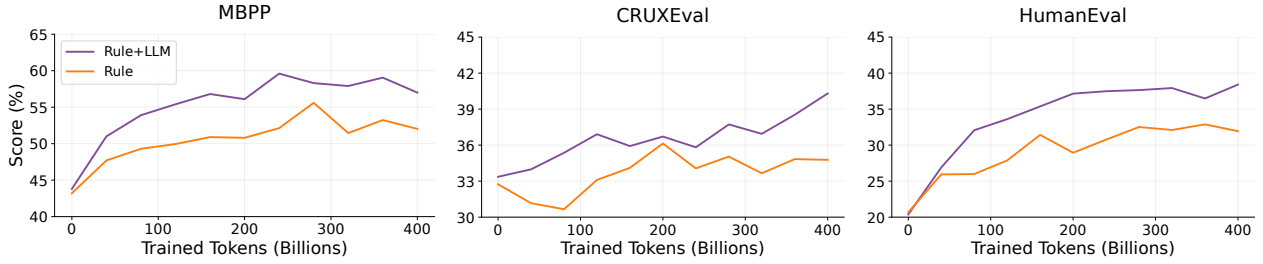
Figure 5. Pipeline of our data quality scorer.

Figure 6. On-the-fly performance over benchmarks during pretraining, from data with minimal rules only (orange) and LLM-based filters involved (purple).

2.2.2. Commits Data

Besides GitHub code files, GitHub commits data encapsulates the collective wisdom of developers, capturing how code evolves through bug fixes, feature updates, and iterative refinement. To better align Seed-Coder with practical development workflows, we incorporated large-scale GitHub commits data into our pretraining. Specifically, we collected 74 million commits from 140K high-quality repositories, selected based on the following criteria: at least 100 stars, 10 forks, 100 commits, and 100 days of maintenance activity. Each commit includes metadata such as the commit message, patch, merge status, and pre-commit code snapshot.

To utilize GitHub commits data for pretraining, we format each sample as a code change prediction task: given a commit message and its associated context, the model predicts the modified file paths and the corresponding code changes. The context includes the pre-commit code snapshot's README, directory structure, and top 5 relevant files retrieved via BM25 [Robertson and Zaragoza, 2009]. After deduplication and preprocessing, we obtained a curated corpus of approximately 100 billion tokens of commits for pretraining, providing dense supervision for learning real-world code change patterns.

2.2.3. Code-Related Web Data

In this section, we propose an optimized framework for extracting code-related data from extensive web archives like Common Crawl[1]. As shown in Figure 7, our extraction framework incorporates
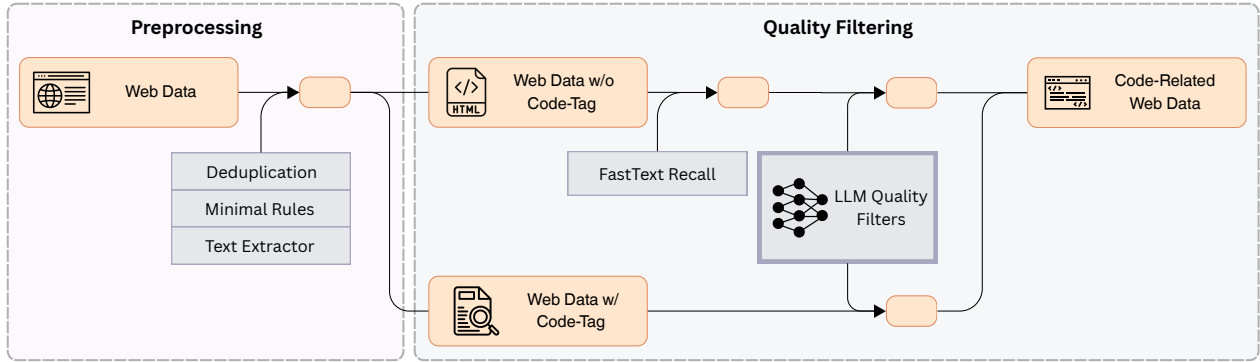
---

[1] https://commoncrawl.org/

Figure 7. Pipeline of extracting high-quality code-related data from web archives.

preprocessing and quality filtering mechanisms, thereby ensuring the harvested code-related data exhibits both high quality and diversity.

Preprocessing. The framework commences with efficient preprocessing of large-scale web archives. We implemented text extraction procedures on Common Crawl and identified two distinct categories of raw data: 1) web pages with explicit code tags (such as <code>...</code>) in HTML that are readily extractable using standard rules, and 2) non-explicit code-tag data potentially containing code or related knowledge, which presents extraction challenges due to its volume and complexity. Paralleling our GitHub data processing approach, we implemented exact- and near-deduplication techniques, and developed heuristics to eliminate low-quality documents (e.g., less than 10 words) during the preprocessing stage.

Quality Filtering. The framework implements dual complementary strategies to ensure data quality: first, identifying content with code relevance, and second, assessing the intrinsic quality of the identified content.

- **FastText Recall:** To handle the complexity and scale of non-explicit code-tag data, we extracted and scored 10 million candidate web pages from Common Crawl data to establish gold-standard datasets for evaluation. From the annotated dataset, we reserved 70% data as the initial seed corpus while preserving the remainder for validation. Using the seed corpus, we trained a fastText model [Joulin et al., 2016, 2017] designed to identify and retrieve additional code-related content. In validation dataset, our model demonstrated a recall rate of 99% alongside a precision rate of 45%. Through this methodology, we successfully identified approximately 3% of Common Crawl data as code-related candidates.

- **LLM Quality Filters:** We employed a scoring system ranging from 0 to 10 to assess the quality of code-related files using an LLM-based evaluator. During our analysis of the collected code-related web content, we observed significant variations in quality scores across different website categories. This variation revealed biases in our initial scoring methodology that required careful mitigation to ensure a balanced and representative dataset. Notably, e-commerce platforms, color-related content, and documentation sites consistently received higher scores due to their standardized formatting and structured presentation, while forums and community discussion platforms typically scored lower despite containing valuable content, primarily attributable to their heterogeneous and less structured formats. To rectify this imbalance and enhance dataset

10

representativeness, we implemented category-specific filtering protocols that adjusted quality thresholds and sampling rates for each category, thereby preventing over representation of individual content types.

The pipeline generated a robust corpus of approximately 1.2 trillion tokens, ensuring balanced representation across diverse web content categories while maintaining both quality standards and categorical diversity.

2.2.4. High-Quality Data for Continued Pretraining

To further improve model performance and better align the distribution between the pretraining model and post-training data, we constructed high-quality datasets by combining the quality score with iteratively trained fastText [Joulin et al., 2016, 2017] models.

These high-quality datasets were obtained from multiple sources, including major programming languages, algorithms, application development, Jupyter notebooks, and general code data. From each source, we initially curated a small yet diverse high-quality seed dataset (approximately 100K samples), selected based on specific data characteristics such as quality scores, programming language, comment ratios, imported packages, etc. This seed dataset served as positive samples for training a fastText model. Negative samples consisted of two parts: randomly selected samples and carefully constructed hard negative samples. Random negative samples were obtained by random selection from the original data after removing the high-quality seeds. Hard negative samples were deliberately designed to resemble high-quality seeds closely, such as code samples with high filter scores but lacking comments or docstrings, or data recalled by the first-round fastText model but scoring poorly according to the quality filter. Hard negative samples were crucial for effectively training the fastText model, as they prevented the model from overfitting positive examples, which otherwise might lead to uniformly high scores and reduced discriminative capability.

Finally, after training 2–3 rounds, each round expanded the seed dataset by incorporating newly identified positive samples, we obtained an effective high-quality data recall model and constructed approximately 130 billion tokens of high-quality data for continued pretraining.

2.2.5. Long-Context Data for Continued Pretraining

We introduce long-context training by supporting sequences of up to 32K tokens during the continued pretraining phase. We conducted two stages, progressively expanding the context window from the original 8K to 32K. This stage utilized approximately 1 trillion training tokens. Training data were organized into two complementary granularities:

- **File-level:** Long context data filtered from GitHub repositories and code-related web data using LLM filtering techniques.

- **Repository-level:** We selected high-quality repositories based on average file quality scores. For mainstream programming languages (e.g., Python, Java, and C), we implemented topological concatenation based on file dependencies. For HTML, SQL, and Shell, we used random concatenation. Each repository was mapped to a single string sequence, with exceptionally large repositories (e.g., PyTorch) being decomposed into multiple independent subgraphs to avoid oversized sequences while preserving logical coherence.

2.2.6. *Fill-in-the-Middle (FIM)*

Beyond the objective of next-token prediction, Bavarian et al. [2022b] introduces the Fill-in-the-Middle (FIM) training to enhance context-aware completion. Due to the flexibility of prompt reformatting in real-world cases, the Suffix-Prefix-Middle (SPM) and Prefix-Suffix-Middle (PSM) modes are considered contextually equivalent. Consequently, most recent works [Hui et al., 2024, DeepSeek-AI et al., 2024b] favor a single-mode FIM approach in their training strategies. However, in contrast to the more commonly adopted PSM mode, we found that SPM performs slightly better during the training process, which is consistent with [Bavarian et al., 2022a]. One possible explanation for this outperformance is the positional bias in attention mechanisms [Yu et al., 2024, Hsieh et al., 2024]: in SPM mode, the beginning of the suffix and the end of the prefix are positioned at opposite ends of the input token sequence, both crucial for middle-content prediction.

During pretraining, we employed character-level random splitting to support intra-word completion. Each FIM sample was converted from the original single-file document into the following format:

<[fim-suffix]>*SUFFIX*<[fim-prefix]>*PREFIX*<[fim-middle]>*MIDDLE*

where <[fim-suffix]>, <[fim-prefix]> and <[fim-middle]> were added as special tokens, and *SUFFIX*, *PREFIX*, *MIDDLE* were the corresponding parts split from the document. For the regular pretraining phase, the FIM ratio was set to 0.5. For the continued pretraining phase, the FIM ratio was set to 0.1.

2.3. Pretraining Policy

Our pretraining model architecture follows the widely adopted Llama 3 [Llama Team, 2024a] structure and is configured as follows: the model has 8.2 billion parameters, 36 layers, with a hidden size of 4,096 and an intermediate size of 14,336, and employs Grouped Query Attention (GQA) with 32 heads for queries and 8 heads for keys and values. We do not apply tied embeddings. The context length during regular pretraining was set to 8K tokens, which was later expanded to 32K tokens during the continued pretraining and post-training phases.

Our pretraining consumed 6 trillion tokens in total. Initially, we pretrained the model with a learning rate of 3e−4 on a mixture of code-related web data and math-related web data for the first 1 trillion tokens. This was followed by an additional 4 trillion tokens of training on curated code data. During the continued pretraining phase, we switched to high-quality and long-context datasets: the learning rate was first reduced by a factor of $\sqrt{10}$, and the model was trained for 400 billion tokens. Subsequently, the learning rate was further decreased to 3e−5, and continued training for an additional 600 billion tokens.

3. Post-training

3.1. Instruct Model

To build Seed-Coder-8B-Instruct, we first applied supervised fine-tuning (SFT) to adapt our base models to a wide range of real-world tasks, followed by direct preference optimization (DPO; Rafailov et al. [2023]) to further enhance specific capabilities such as code generation and reasoning.
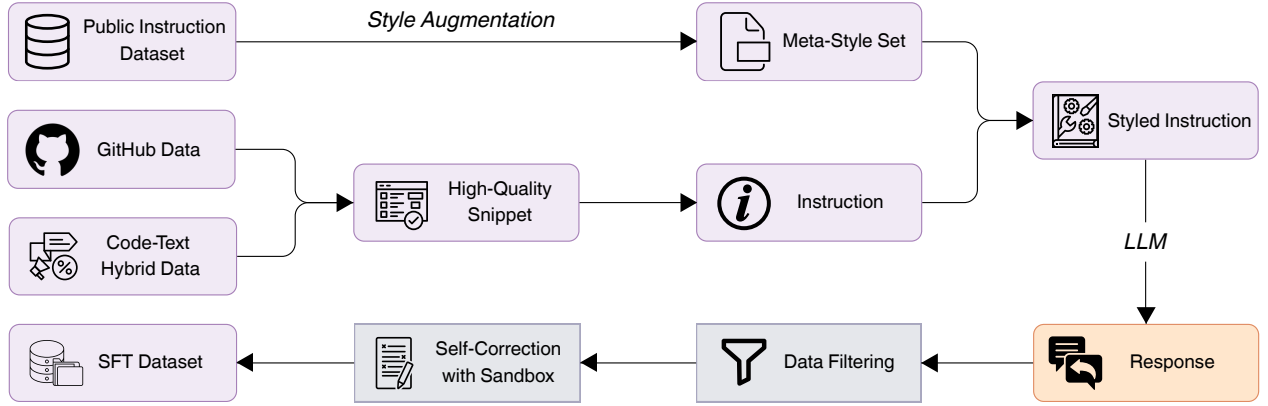
Figure 8. Pipeline of synthetic data curation for our instruct model training.

For SFT, we constructed an instruction tuning dataset containing millions of samples. In curating this dataset, we focused on three primary aspects: diversity, quality, and difficulty, which are critical for improving the model's robustness and generalization. The pipeline for SFT dataset curation is illustrated in Figure 8. Specifically, we first collected high-quality code snippets from curated GitHub repositories and code-text hybrid datasets, and built a meta-style set through style augmentation over public instruction data. These sources were then used to prompt LLMs to synthesize diverse, styled instructions [Wei et al., 2024] and corresponding responses. The instruction-response pairs were filtered through a combination of rule-based and model-based techniques to ensure high quality and appropriate difficulty levels. Finally, we applied a sandbox-based self-correction mechanism to iteratively refine the outputs. The finalized high-quality examples were compiled into the SFT dataset for training. We detail this process in the following subsections.

3.1.1. Data Construction: Diversity

We primarily employed synthetic data generation to construct an instruction dataset emphasizing diversity, difficulty, scalability, and quality. During prompt synthesis, we prioritized prompt diversity to enhance the LLM's robustness across diverse scenarios and edge cases. We leveraged high-quality seed snippets to promote diversity in two dimensions: seed snippet diversity and style diversity.

Seed Snippet Diversity. To ensure broad coverage, we collected code snippets from multiple high-quality sources. Specifically, we extracted curated GitHub code snippets, applying rigorous filtering based on file quality scores. We then utilized OSS-Instruct [Wei et al., 2024] to efficiently generate additional code snippets. While isolated code snippets provide valuable supervision signals, they may not sufficiently expose the model to complex real-world scenarios and user interaction patterns. To address this, we expanded our data sources to include code-text hybrid datasets such as Markdown files, Jupyter notebooks, and StackExchange discussions. These sources offer not only executable codes but also accompanying explanations, usage examples, and problem-solving discussions; thus, they better approximate real-world user interactions. By incorporating both codes and surrounding contexts, we enable the model to generate instructions that closely align with how developers write, explain, and seek assistance.

Style Diversity. Although synthetic data offers an effective and cost-efficient alternative to real-

13

world data [Liu et al., 2024a], it often lacks the stylistic diversity observed in actual user prompts. For instance, model-generated instructions are typically well-structured and complete, whereas real-world prompts tend to be more casual and varied. To bridge this gap, we built a *meta-style set* by collecting diverse instructions from public datasets. To further enrich this set, we applied *style augmentation*, wherein two styles were randomly selected and blended to synthesize new styles. During instruction generation, prompts were reformatted to match randomly selected styles from the meta-style set, thereby enhancing the model's robustness against different prompting behaviors.

To further improve the model's ability to respond to real-world queries, we additionally incorporated code-related data sampled from WildChat [Zhao et al., 2024] into our SFT dataset.

3.1.2. Data Filtering: Quality and Difficulty

We employed a combination of rule-based and model-based techniques to filter out low-quality SFT pairs from the dataset.

Quality Filtering. To ensure data quality, we applied both rule-based and model-based filters. For rule-based filtering, we used Tree-sitter [Brunsfeld, 2018] to eliminate responses containing syntax errors. For model-based filtering, we prompted an evaluation model to score responses based solely on correctness, discarding those with low scores.

Difficulty Filtering. We first classified the topic of each instance using tagging techniques [Lu et al., 2023]. Subsequently, we prompted a model to assess the difficulty level of each instance within its domain. Instances receiving a difficulty score lower than 3 out of 10 were discarded from the dataset.

3.1.3. Self-Correction with Sandbox Verification

In our experiments, we observed that examples with higher difficulty scores often exhibit higher error rates, resulting in many challenging instances being filtered out during quality filtering. To mitigate this and recall more high-difficulty examples in the SFT dataset, we implemented a sandbox verification and self-correction framework. Specifically, we prompted the model to generate solutions along with corresponding unit tests, evaluate the outputs within a sandbox environment, and iteratively refine any failed solutions until all tests passed or a maximum number of revision attempts was reached. This iterative refinement process allows us to retain more challenging examples and enrich the difficulty distribution of the training corpus.

3.1.4. Direct Preference Optimization

To further enhance the model's capabilities in code generation and reasoning, we constructed on-policy preference data for DPO. We first selected task-relevant prompts and sampled hundreds of candidate responses for each. These responses were then evaluated in a sandbox environment using generated code and corresponding unit tests. Based on the evaluation results, we formed preference pairs for DPO training to refine the model's behavior.

3.1.5. Training Recipe for Instruct Model

We fine-tuned Seed-Coder-8B-Instruct in two stages: In the first stage, we performed SFT on the instruction dataset described above. The dataset contained approximately 3 million high-quality instruction-response pairs spanning diverse tasks and domains. To enhance the model's ability to handle challenging tasks, we employed difficulty-aware sampling to prioritize higher-difficulty examples during training. All prompts were reformatted using a system message and a chat-style template to better align the model's outputs with instruction-following conventions. The model was trained for 3 epochs using a learning rate of 2e−5 and enabled sequence packing to improve training efficiency. In the second stage, we applied DPO to further strengthen the model's capabilities in code generation and reasoning. We constructed approximately 20,000 high-quality preference pairs by focusing on challenging examples from these domains.

This two-stage training strategy enables the model to learn both general instruction-following behaviors and fine-grained preferences in complex domains, ultimately leading to improved robustness, reasoning ability, and code generation performance.

3.2. Reasoning Model

Recently, models employing Long-Chain-of-Thought (LongCoT) reasoning have demonstrated significant performance gains on complex and verifiable tasks, such as mathematics and coding problems. To explore the upper limits of our 8B-sized base model, we conducted LongCoT reinforcement learning (RL) training and evaluated the model's performance on difficult problems. Starting from our base model, we first performed a LongCoT warmup phase, followed by GRPO-based RL training to further enhance its capabilities.

3.2.1. Data

We collected challenging real-world coding problems, including CodeContests [Li et al., 2022] and ICPC problems[2], and gathered model-generated solutions from DeepSeek-R1 [DeepSeek-AI et al., 2025] on these tasks. Additionally, we incorporated open-source chain-of-thought datasets such as open-r1/codeforces-cots [Penedo et al., 2025] for a distilled model. Note that, for CodeContests and ICPC data, we applied sandbox-based rejection sampling during collection, retaining only correct model generations. This ensures that our warmup set provides stronger supervision compared to other distilled models. For the RL training, we utilized the aforementioned datasets combined with LiveCodeBench data collected prior to August 2024, excluding any unverifiable samples.

3.2.2. Warmup Step

Before proceeding to reinforcement learning (RL) training, we fine-tuned Seed-Coder-8B-Base on thousands of collected warmup LongCoT samples using a learning rate of 2e−5. We initiated the warmup process from the base model rather than the instruct model, despite the latter achieving higher scores immediately following warmup completion. This decision stemmed from our observation that the instruct model frequently collapses into typical SFT training data patterns during RL, ultimately degrading its post-RL performance.

---

[2]https://icpc.global/worldfinals/past-problems

During the warmup phase, we observed that model performance improvements were positively correlated with the amount of distillation data used. However, as pointed out by Yue et al. [2025], a large volume of distillation data may exceed the inherent capabilities of the base model and thus fail to accurately reflect its true potential. To preserve the model's own exploration space, we chose not to further scale up the distillation data. Instead, we employed a relatively small amount of data to encourage the base model to learn diverse thinking patterns, relying on RL for subsequent self-improvement in order to better explore the model's upper bound.

3.2.3. Training Recipe for Reasoning Model

During the RL training process, similar to DeepCoder [Luo et al., 2025], we used the open-source verl framework [Sheng et al., 2025] for GRPO training [Shao et al., 2024], and adopted optimization techniques similar to DAPO [Yu et al., 2025, ByteDance Seed, 2025]. The training configuration included a batch size of 128, an initial learning rate of 1e−6, a temperature of 0.6, and we removed the KL loss. We set the clip ratio high at 0.28, filtered overlong samples, and applied token-wise loss.
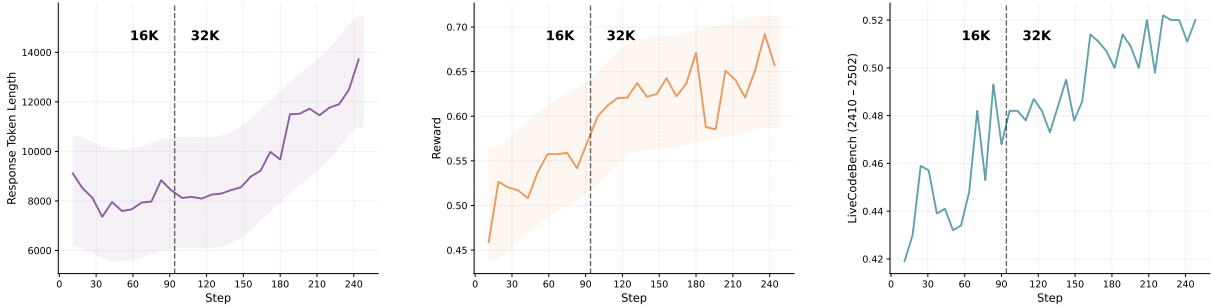
Figure 9. RL training trajectories: smoothed average response token length (left), smoothed average rewards (middle), LiveCodeBench pass@1 performance (right). The area to the left of the dashed line corresponds to a 16K window, and the area to the right corresponds to a 32K window.

In addition, based on several empirical observations, we applied further modifications:

Optimized Curriculum Learning. While GRPO inherently incorporates curriculum learning by filtering entirely correct or incorrect samples, we identified two areas for improvement: For simple problems, the model's thinking process contained significant redundancy, including repetitive examples and superficial logic. Since thinking patterns play a crucial role in scaling capabilities, we further filtered simple problems – those with correctness rates above 87.5% – to encourage more efficient utilization of thinking tokens. Moreover, due to the addition of format rewards, situations arose with groups containing positive-only examples and format errors, which would not be automatically filtered out by GRPO. As these also qualify as simple problems, we filtered them out as well.

Progressive Exploration Strategy. We adopted a gradually expanding strategy for sequence length and rollout number during training, further extending the progressive approach introduced in [Luo et al., 2025]. Specifically, we first trained for 90 steps with a 16K sequence length and 16 samples per prompt, followed by 160 steps with a 32K sequence length and 32 samples per prompt, totaling 250 steps. Through empirical comparison, we found that expanding the sequence length and the rollout

number yielded comparable final results, but the progressive schedule significantly improved training efficiency during the early stages. Preliminary observations also suggest that further increasing the rollout size to 64 or higher could potentially lead to even better performance. We encourage future work to explore more systematic strategies for allocating computational resources across different learning stages.

4. Decontamination

To ensure that our training data is not affected by potential test data leakage, we performed a decontamination process on the entire training data, including our pretraining and post-training data. Specifically, we employed the widely adopted 10-gram filtering method [Guo et al., 2024, Hui et al., 2024], which removed all data that had any 10-gram word overlap with key benchmark datasets, including HumanEval, MBPP, MHPP, BigCodeBench, LiveCodeBench, Aider, FullStack Bench, etc.

5. Results

We performed extensive series evaluations to investigate the performance of 1) Seed-Coder-8B-Base: the pretrained base model, 2) Seed-Coder-8B-Instruct: the instruction fine-tuned model, and 3) Seed-Coder-8B-Reasoning: the reasoning model trained by reinforcement learning. We present the results of these evaluations in separate sections below. We compare with previous state-of-the-art open-source LLMs as follows.

- **StarCoder2** [Lozhkov et al., 2024] is the successor of StarCoder [Li et al., 2023], a classic code LLM with 15.5B parameters trained on 1 trillion tokens sourced from the Stack [Li et al., 2023], which consists of 6.4TB of permissively licensed source code in 384 programming languages. The fully upgraded version StarCoder2 consists of 3B, 7B, and 15B parameters models trained on 3.3 to 4.3 trillion tokens of the Stack v2 [Lozhkov et al., 2024], with 67.5TB code data spanning 619 programming languages.

- **Codestral** [Mistral AI, 2024] is an open-weight 22B parameter model released by Mistral AI, which is explicitly designed for code generation tasks. It is trained on a dataset of 80+ programming languages.

- **CodeLlama** [Rozière et al., 2024] is a family of code LLMs developed based on Llama 2 [Touvron et al., 2023] models, by firstly training on 500B to 1T tokens from a code-heavy dataset and then fine-tuning with additional 5B tokens to better follow human instructions. CodeLlama is available in four sizes: 7B, 13B, 34B, and 70B parameters.

- **DeepSeek-Coder** [Guo et al., 2024] is a series of code LLMs with 1.3B, 6.7B, and 33B parameters, each trained from scratch on 2 trillion tokens, with a composition of 87% code and 13% natural language in both English and Chinese. These models are pretrained on a project-level code corpus and an additional fill-in-the-blank task, which enables project-level code completion and infilling.

- **CodeQwen1.5** [Qwen Team, 2024a] is a specialized code LLM with 7B parameters built upon the Qwen1.5 language model [Qwen Team, 2024b], via training on around 3 trillion tokens of code-related data that covers 92 programming languages.

- **Yi-Coder** [01.AI, 2024] is a series of code LLMs in two sizes (1.5B and 9B), trained on top of 2.4 trillion high-quality tokens sourced from a repository-level GitHub code corpus that covers 52 programming languages and code-related data filtered from CommonCrawl.

- **Llama 3.1** [Llama Team, 2024b] is a suite of general LLMs with 8B, 70B, and 405B model parameters, which is an upgrade of the previously released Llama 3 [Llama Team, 2024a] series. Llama 3.1 is pretrained on a corpus of about 15 trillion multilingual tokens, compared to 1.8 trillion tokens for its predecessor Llama 2 [Touvron et al., 2023].

- **OpenCoder** [Huang et al., 2025] is a series of code LLMs which includes 1.5B and 8B parameter models, trained from scratch on 2.5 trillion tokens composed of 90% raw code and 10% code-related web data.

- **DeepSeek-Coder-V2** [Guo et al., 2024] is a series of Mixture-of-Experts (MoE) code LLMs with 16B and 236B parameters, which has 2.4B and 21B activation parameters, respectively. These models are further pretrained from a checkpoint at 4.2 trillion tokens of DeepSeek-V2 [DeepSeek-AI et al., 2024b] with additional 6 trillion tokens comprising 60% source code, 10% math corpus, and 30% natural language corpus.

- **Qwen2.5-Coder** [Hui et al., 2024] is a series of code LLMs that has a rich variety of model sizes, including 0.5B, 1.5B, 3B, 7B, 14B, and 32B parameter models, which are further pretrained from Qwen2.5 [Qwen Team, 2024c] on over 5.5 trillion tokens of code-related data. As the flagship model of this series, Qwen2.5-Coder-32B-Instruct has achieved the best overall performance among open-source models on various coding benchmarks, and even has competitive performance with powerful proprietary LLMs such as GPT-4o.

- **Qwen3** [Qwen Team, 2025] is the most recent addition to the Qwen family of open-source LLMs. The Qwen3 series includes two MoE models – Qwen3-235B-A22B (235B total parameters, with 22B activated parameters) and Qwen3-30B-A3B (30B total parameters, with 3B activated parameters) – as well as six dense models: Qwen3-32B, Qwen3-14B, Qwen3-8B, Qwen3-4B, Qwen3-1.7B, and Qwen3-0.6B. The Qwen3 models introduce a hybrid approach consisting of a "non-thinking mode" and a "thinking mode": In non-thinking mode, the model provides quick responses, whereas in thinking mode, the model reasons in a step-by-step manner before generating a final response. To ensure fair evaluation that accounts for both model size and modeling type, we compare the performance of Qwen3-8B in non-thinking mode against standard instruct models, and its performance in thinking mode against dedicated reasoning models.

5.1. Evaluation of Base Models

In this section, we report the evaluation results of our pretrained base model Seed-Coder-8B-Base, compared with various previous state-of-the-art models to assess their capabilities in code generation, code completion, and code reasoning. To ensure the reproducibility of our results, we use the open-source evaluation suites released by Qwen2.5-Coder [Hui et al., 2024], adopting their reported performance metrics when available and conducting our own evaluations for the remaining ones.

Model	Size	HumanEval		MBPP	
		HE	*HE*+	*MBPP*	*MBPP*+
~8B Models					
StarCoder2-7B	7B	35.4	29.9	54.4	45.6
DeepSeek-Coder-6.7B-Base	6.7B	47.6	39.6	70.2	56.6
CodeQwen1.5-7B	7B	51.8	45.7	72.2	60.2
OpenCoder-8B-Base	8B	66.5	63.4	79.9	**70.4**
Qwen2.5-Coder-7B	7B	72.0	67.1	79.4	68.3
Seed-Coder-8B-Base	8B	**77.4**	**68.3**	**82.0**	69.0
13B+ Models					
StarCoder2-15B	15B	46.3	37.8	66.2	53.1
CodeLlama-70B-Base	70B	52.4	50.6	71.0	65.6
Llama-3.1-70B-Base	70B	54.9	51.2	81.4	67.2
DeepSeek-Coder-33B-Base	33B	54.9	47.6	74.2	60.7
DeepSeek-Coder-V2-Lite-Base	2.4B/16B	40.9	34.1	71.9	59.4
Qwen2.5-Coder-14B	14B	**83.5**	**75.6**	**83.6**	**69.8**

Table 1. Performance of various base models on HumanEval(+) and MBPP(+).

5.1.1. Code Generation

HumanEval [Chen et al., 2021] consists of 164 manually written programming tasks in Python, each of which provides a function signature with a docstring for LLMs to complete a solution, whose functional correctness is judged by a handful of test cases. To rigorously benchmark the correctness of LLM-generated code, EvalPlus [Liu et al., 2023] extends the test cases of the original HumanEval by 80× to build HumanEval+. We used EvalPlus to evaluate base models on both HumanEval and HumanEval+.

MBPP [Austin et al., 2021] is created by crowd-sourcing participants to write 974 Python programming problems, each of which is comprised of a function signature with a docstring, as well as a few test cases. We adopt the human-verified version of MBPP in EvalPlus, which is a subset of 399 tasks that are verified to be well-formed. Similarly, to accurately reflect the true performance of LLMs, we also evaluated base models on MBPP+ with 35× more test cases than the original MBPP.

The evaluation results of HumanEval(+) and MBPP(+) are presented in Table 1. When using EvalPlus to evaluate Qwen2.5-Coder series, we found that although these models were supposed to be base models, they came with an official chat template and their performance on HumanEval and MBPP would be significantly boosted when the chat template was applied. Without the chat template applied, EvalPlus reports similar base model performance as in the Qwen2.5-Coder report [Hui et al., 2024]. To reflect the genuine ability of Qwen2.5-Coder series, we decide to report their higher EvalPlus scores with the chat template applied. For Seed-Coder-8B-Base, which has no chat template, we report its EvalPlus scores under the normal base prompt setting. As shown in Table 1, Seed-Coder-8B-Base achieves impressive performance among open-source models of similar size, and even surpasses some much larger models.

MultiPL-E. Besides Python, we also evaluated base models on their code generation abilities across other popular programming languages. MultiPL-E [Cassano et al., 2023] extends HumanEval

19

Model	Size	Python	C++	Java	PHP	TS	C#	Bash	JS	Average
~8B Models										
StarCoder2-7B	7B	35.4	40.4	38.0	30.4	34.0	46.2	13.9	36.0	34.3
DeepSeek-Coder-6.7B-Base	6.7B	49.4	50.3	43.0	38.5	49.7	50.0	28.5	48.4	44.7
CodeQwen1.5-7B	7B	51.8	52.2	42.4	46.6	52.2	55.7	36.7	49.7	48.4
OpenCoder-8B-Base	8B	66.5	63.4	63.9	61.5	68.6	54.3	44.3	65.8	61.0
Qwen2.5-Coder-7B	7B	72.0	62.1	53.2	59.0	64.2	**60.8**	38.6	60.3	58.8
Seed-Coder-8B-Base	8B	**77.4**	**69.6**	**72.8**	**63.9**	**77.4**	53.8	**48.1**	**77.6**	**67.6**
13B+ Models										
StarCoder2-15B	15B	46.3	47.2	46.2	39.1	42.1	53.2	15.8	43.5	41.7
CodeLlama-70B-Base	70B	52.4	49.7	44.7	46.6	57.2	46.7	31.6	56.5	48.2
Llama-3.1-70B-Base	70B	54.9	41.0	41.1	48.4	57.9	44.2	29.1	55.3	46.5
DeepSeek-Coder-33B-Base	33B	56.1	58.4	**51.9**	44.1	52.8	51.3	32.3	55.3	50.3
DeepSeek-Coder-V2-Lite-Base	2.4B/16B	40.9	45.9	34.8	47.2	48.4	41.7	19.6	44.7	40.4
Qwen2.5-Coder-14B	14B	**83.5**	**69.6**	46.8	**64.6**	**69.2**	**63.3**	**39.9**	61.5	**62.3**

Table 2. Performance of base models on MultiPL-E.

by translating Python tasks and unit tests into 18 additional programming languages, including C++, Java, PHP, TypeScript (TS), C#, Bash, JavaScript (JS), etc. Evaluating models on MultiPL-E provides a comprehensive overview of how they work across various languages and helps understand the language factors that may affect the coding performance of LLMs. Following Qwen2.5-Coder [Hui et al., 2024], we chose eight mainstream programming languages from MultiPL-E for evaluation and present the results in Table 2. On 7 out of 8 programming languages, Seed-Coder-8B-Base outperforms other open-source models of comparable size, and on some languages, even outperforms models with over 13B parameters. This shows the superior multilingual code generation abilities of Seed-Coder-8B-Base.

5.1.2. Code Completion

Code completion is one of the most essential features that code LLMs provide to developers, enabling the suggestion of code snippets based on both preceding and succeeding context. We evaluated the code completion performance of base models using three benchmarks – CrossCodeEval, RepoEval and Single-line MultiPL-HumanEval FIM – each assessed under the FIM format.

CrossCodeEval [Ding et al., 2023] is composed of single-line infilling tasks across Python, TypeScript, C# and Java. In addition to the prefix and suffix context in the same file as the reference, it provides cross-file-related codes to create a self-contained code snippet. For evaluation parameters, we set the maximum sequence length to 8,192 tokens, limited the maximum generated length to 50 tokens, and added cross-file context of no more than 2,048 tokens, computed under Retrieval w/ Ref. mode as in the original paper of CrossCodeEval. Table 3 summarizes the performance of base models under the Exact Match (EM) and Edit Similarity (ES) metrics. The average is an unweighted mean of language-wise results.

RepoEval [Zhang et al., 2023] features API invocation and function-body completion alongside single-line completion tasks. It is a Python-oriented benchmark reorganized from selected high-quality GitHub repositories. We set the maximum sequence length to 8,192 tokens and the maximum cross-file context length to 2,048 tokens for all evaluation tasks. For API invocation and single-line

Model	Size	Python		Java		TypeScript		C#		Average	
		EM	ES	EM	ES	EM	ES	EM	ES	EM	ES
~8B Models											
StarCoder2-7B	7B	10.9	63.1	8.3	71.0	6.7	76.8	7.3	72.1	8.3	70.8
DeepSeek-Coder-6.7B-Base	6.7B	41.1	79.2	39.9	80.1	46.3	82.4	55.0	86.9	45.6	82.1
CodeQwen1.5-7B	7B	40.7	77.8	47.0	81.6	45.8	82.2	59.7	87.6	48.3	82.3
Qwen2.5-Coder-7B	7B	42.4	78.6	48.1	82.6	46.8	83.4	59.7	87.9	49.3	83.1
Seed-Coder-8B-Base	8B	**49.5**	**82.0**	**52.6**	**84.5**	**50.5**	**84.6**	**62.0**	**89.1**	**53.7**	**85.1**
13B+ Models											
StarCoder2-15B	15B	28.2	70.5	26.7	71.0	24.7	76.3	25.2	74.2	26.2	73.0
DeepSeek-Coder-33B-Base	13B	44.2	80.4	46.5	82.7	49.2	84.0	55.2	87.8	48.8	83.7
DeepSeek-Coder-V2-Lite-Base	2.4B/16B	41.8	78.3	46.1	81.2	44.6	81.4	58.7	87.9	47.8	82.2
Qwen2.5-Coder-14B	14B	47.7	81.7	54.7	85.7	52.9	86.0	66.4	91.1	55.4	86.1
Qwen2.5-Coder-32B	32B	**49.2**	**82.1**	**56.4**	**86.6**	**54.9**	**87.0**	**68.0**	**91.6**	**57.1**	**86.8**

Table 3. Performance of base models on CrossCodeEval.

Model	Size	Line		Function		API		Average	
		EM	ES	EM	ES	EM	ES	EM	ES
~8B Models									
StarCoder2-7B	7B	19.5	67.6	4.0	53.5	19.1	72.8	14.2	64.7
DeepSeek-Coder-6.7B-Base	6.7B	63.1	85.5	9.9	53.3	52.3	81.7	41.7	73.5
CodeQwen1.5-7B	7B	59.7	81.5	4.8	44.3	46.1	77.5	36.9	67.8
Qwen2.5-Coder-7B	7B	67.3	86.1	13.2	**55.2**	58.4	83.9	46.3	75.1
Seed-Coder-8B-Base	8B	**72.5**	**88.7**	**19.3**	53.0	**60.8**	**85.0**	**50.8**	**75.6**
13B+ Models									
StarCoder2-15B	15B	30.9	62.5	5.5	43.7	21.7	60.3	19.4	55.5
DeepSeek-Coder-33B-Base	13B	66.5	86.6	10.3	52.9	54.2	83.5	43.7	74.3
DeepSeek-Coder-V2-Lite-Base	2.4B/16B	66.5	85.4	10.8	53.9	53.1	81.3	43.4	73.5
Qwen2.5-Coder-14B	14B	74.3	90.1	**14.1**	**59.5**	63.4	87.3	50.6	**79.0**
Qwen2.5-Coder-32B	32B	**76.1**	**90.5**	13.6	57.5	**65.1**	**87.6**	**51.6**	78.5

Table 4. Performance of base models on RepoEval.

completion, the number of predicted tokens was limited to 50, and for function-body completion, 256. Cross-file context was retrieved with the oracle method aligned to the original paper. The performance of base models on RepoEval is shown in Table 4. EM and ES remain as the metrics, and the average is computed with uniform weights from language-wise results.

Single-line MultiPL-HumanEval FIM [Allal et al., 2023] refines MultiPL-E [Cassano et al., 2023] into a set of single-line infilling challenges across Python, Java, and JavaScript, which serves as a classic benchmark for FIM evaluation. Performance is assessed using the EM metric [Fried et al., 2023]. Table 5 presents the results on Single-line MultiPL-HumanEval FIM, comparing the models by size. The average is weighted by the number of samples from each language.

Although the EM and ES metrics may not fully capture true performance since semantically equiva-

Model	Size	Python	Java	JavaScript	Average
		~8B Models			
StarCoder2-7B	7B	70.8	86.0	84.4	82.0
DeepSeek-Coder-6.7B-Base	6.7B	78.1	87.4	84.1	84.0
CodeQwen1.5-7B	7B	75.8	85.7	85.0	83.3
Qwen2.5-Coder-7B	7B	**79.7**	**88.5**	**87.6**	**86.2**
Seed-Coder-8B-Base	8B	77.1	84.5	83.2	82.4
		13B+ Models			
StarCoder2-15B	15B	74.2	85.2	84.6	82.6
DeepSeek-Coder-33B-Base	33B	80.1	89.0	86.8	86.2
DeepSeek-Coder-V2-Lite-Base	2.4B/16B	78.7	87.8	85.9	85.0
Qwen2.5-Coder-14B	14B	80.5	**91.0**	88.5	87.7
Qwen2.5-Coder-32B	32B	**81.5**	**91.0**	**89.4**	**88.3**

Table 5. Exact Match scores of base models on the task of single-line MultiPL-HumanEval FIM.

Model	Size	Input-CoT	Output-CoT
		~8B Models	
StarCoder2-7B	7B	39.5	35.1
DeepSeek-Coder-6.7B-Base	6.7B	39.0	41.0
CodeQwen1.5-7B	7B	44.8	40.1
OpenCoder-8B-Base	8B	43.3	43.9
Qwen2.5-Coder-7B	7B	**56.5**	**56.0**
Seed-Coder-8B-Base	8B	52.0	54.8
		13B+ Models	
StarCoder2-15B	15B	46.1	47.6
CodeLlama-34B-Base	34B	49.4	43.9
DeepSeek-Coder-33B-Base	33B	50.6	48.8
DeepSeek-Coder-V2-Lite-Base	2.4B/16B	53.4	46.1
Qwen2.5-Coder-14B	14B	**60.6**	**66.4**

Table 6. Performance of base models on CRUXEval.

lent statements can be expressed in various ways, Seed-Coder-8B-Base demonstrates superiority across the ~8B model family.

5.1.3. Code Reasoning

CRUXEval. Compared to natural language, code is highly sensitive to even minor modifications, including single-token changes. Therefore, effective code reasoning demands precise token-level understanding and the capacity to internally simulate or trace the behavior of code. To evaluate the capability of code reasoning for base models, we utilized the CRUXEval [Gu et al., 2024] benchmark, which comprises 800 Python functions paired with corresponding input-output examples. CRUXEval is divided into two distinct tasks: CRUXEval-I, in which the model predicts the output from a given input, and CRUXEval-O, in which the model infers inputs from provided outputs.
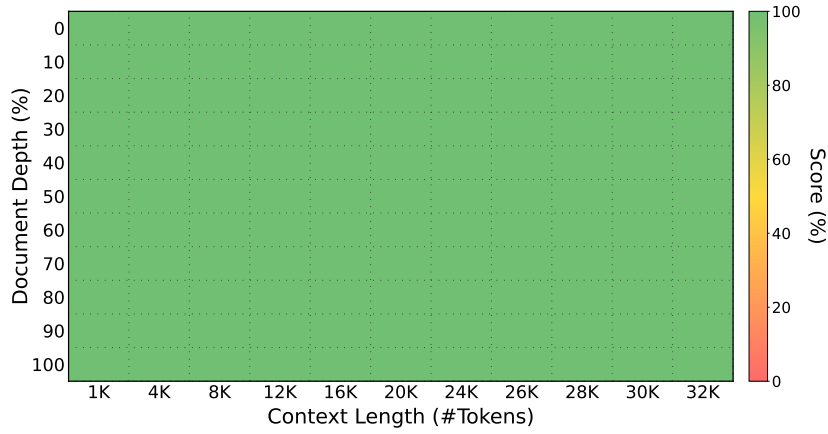
Figure 10. Evaluation results of the "Needle in the Code" pressure test.

This structure tests the model's capability to comprehend and reason about Python code in both forward and backward directions.

Table 6 presents the results for both tasks evaluated in the Chain-of-Thought (CoT) mode, where the model is prompted to reason step by step during simulated code execution. Notably, in both the Input-CoT and Output-CoT settings, Seed-Coder-8B-Instruct achieves top-tier performance among ~8B models, only marginally trailing Qwen2.5-Coder-7B. Note that both our model and Qwen2.5-Coder-7B surpass some much larger models such as DeepSeek-Coder-33B-Base.

5.1.4. Long-Context Capability

Needle in the Code. To unlock the potential of our code model in understanding repository-level code contexts and benefit real-world software development, it is vital to involve long-context capability in the pretraining phase. To this end, we build a "Needle in the Code" pressure test. The needle construction method is a specialized evaluation technique designed to assess LLM's ability to handle long contexts. This approach involves carefully placing a relevant code snippet (the "needle") in broader code functions (the "haystack"). The model is then presented with a specific query related to the embedded code and is required to find and identify the particular code snippet within the larger functional context. Figure 10 shows that Seed-Coder-8B-Base achieves 100% accuracy in the "Needle in the Code" pressure test under 32K context length.

5.2. Evaluation of Instruct Models

In this section, we report comprehensive evaluation results of our instruct model Seed-Coder-8B-Instruct and previous state-of-the-art open-source models on a wide range of coding tasks, including code generation, code reasoning, code editing, and software engineering. To guarantee reproducibility, we utilized the open-source evaluation suites provided by Qwen2.5-Coder [Hui et al., 2024], incorporating their published performance metrics whenever available, and performing our own evaluations for metrics not previously reported.

Model	Size	HumanEval		MBPP		MHPP	BigCodeBench		LiveCodeBench
		HE	HE+	MBPP	MBPP+	pass@1	Full	Hard	2410 – 2502
~8B Models									
CodeLlama-7B-Instruct	7B	40.9	33.5	54.0	44.4	6.7	21.9	3.4	3.6
DeepSeek-Coder-6.7B-Instruct	6.7B	74.4	71.3	74.9	65.6	20.0	35.5	10.1	9.6
CodeQwen1.5-7B-Chat	7B	83.5	78.7	77.7	67.2	17.6	39.6	18.9	3.0
Yi-Coder-9B-Chat	9B	82.3	74.4	82.0	69.0	26.7	38.1	11.5	17.5
Llama-3.1-8B-Instruct	8B	68.3	59.8	70.1	59.0	17.1	36.6	13.5	11.5
OpenCoder-8B-Instruct	8B	83.5	78.7	79.1	69.0	30.5	40.3	16.9	17.1
Qwen2.5-Coder-7B-Instruct	7B	**88.4**	**84.1**	83.5	**71.7**	26.7	41.0	18.2	17.3
Qwen3-8B	8B	84.8	80.5	77.0	67.2	32.8	51.7	23.0	23.5
Seed-Coder-8B-Instruct	8B	84.8	78.7	**85.2**	71.2	**36.2**	**53.3**	**26.4**	**24.7**
13B+ Models									
StarCoder2-15B-Instruct	15B	67.7	60.4	78.0	65.1	19.0	37.2	11.5	5.3
Codestral-22B	22B	81.1	73.2	78.2	62.2	25.2	41.8	16.9	20.5
CodeLlama-70B-Instruct	70B	72.0	65.9	77.8	64.6	19.5	40.7	11.5	14.5
DeepSeek-Coder-33B-Instruct	33B	81.1	75.0	80.4	70.1	32.9	42.0	17.6	14.5
DeepSeek-Coder-V2-Lite-Instruct	2.4B/16B	81.1	75.6	82.8	70.4	30.5	36.8	16.2	14.2
DeepSeek-Coder-V2-Instruct	21B/236B	85.4	82.3	89.4	**75.1**	31.9	48.2	24.3	28.9
Qwen2.5-Coder-14B-Instruct	14B	89.6	**87.2**	86.2	72.8	36.7	48.4	22.2	19.3
Qwen2.5-Coder-32B-Instruct	32B	92.7	**87.2**	**90.2**	**75.1**	**42.4**	**49.6**	**27.0**	**30.7**

Table 7. Performance of instruct models on HumanEval($^+$), MBPP($^+$), MHPP, BigCodeBench-Instruct and LiveCodeBench (2410 – 2502).

5.2.1. Code Generation

HumanEval and MBPP. As introduced in Sec 5.1.1, we also evaluated basic code generation abilities of instruct models on HumanEval($^+$) and MBPP($^+$) using EvalPlus [Liu et al., 2023]. As shown in Table 7, Seed-Coder-8B-Instruct achieves competitive performance on these conventional code generation benchmarks.

MHPP. With the rapid advancement of code LLMs, traditional coding benchmarks such as HumanEval and MBPP are no longer sufficient to effectively differentiate between state-of-the-art models. High overall pass rates on these conventional benchmarks alone cannot reliably indicate whether a model possesses the complex reasoning abilities required to solve difficult coding problems.

Recently, a challenging benchmark named Mostly Hard Python Problems (MHPP) was introduced [Dai et al., 2024], consisting of 210 human-curated problems accompanied by unit tests. MHPP focuses on evaluating LLMs' abilities to tackle various challenges in code generation: handling variance in natural language inputs, understanding newly defined contexts, demonstrating commonsense reasoning, handling edge cases, following intricate instructions, applying mathematical and algorithmic knowledge, and exhibiting familiarity with coding principles. To compare the code generation performance of code LLMs under these challenges, we evaluated various instruct models on MHPP and report the results in Table 7. Our Seed-Coder-8B-Instruct model achieves a striking 36.2% pass@1 score, surpassing all the other ~8B models by a significant margin and even outperforming some much larger models such as DeepSeek-Coder-V2-Instruct with 236B total parameters.

BigCodeBench. To assess how well LLMs can solve challenging and real-world programming

Model	Size	Python	Java	C++	C#	TS	JS	PHP	Go	Kotlin	Perl	Ruby	Scala	Swift	Average
						~8B Models									
CodeLlama-7B-Instruct	7B	54.0	38.8	32.9	50.0	42.3	45.5	36.6	48.8	47.2	50.1	36.9	40.2	33.2	42.8
DeepSeek-Coder-6.7B-Instruct	6.7B	74.9	52.2	30.9	55.9	64.8	64.7	25.8	93.8	59.6	3.3	65.9	54.8	47.4	53.4
CodeQwen1.5-7B-Chat	7B	77.7	66.6	66.8	64.4	66.7	67.5	67.3	55.1	60.9	61.1	65.9	60.0	54.7	64.2
Yi-Coder-9B-Chat	9B	82.0	73.4	79.1	70.3	74.1	73.3	76.4	90.9	64.4	60.9	67.3	63.5	57.3	71.8
Llama-3.1-8B-Instruct	8B	70.1	59.8	59.1	56.6	59.1	59.1	62.5	85.7	52.2	42.6	55.9	44.5	31.8	56.8
OpenCoder-8B-Instruct	8B	79.1	68.1	71.3	71.0	67.6	61.4	68.1	94.4	66.4	56.1	70.5	63.1	56.7	68.8
Qwen2.5-Coder-7B-Instruct	7B	83.5	70.5	74.1	71.5	72.2	74.1	74.2	**96.0**	65.5	64.4	75.5	64.2	**62.0**	72.9
Qwen3-8B	8B	77.0	69.0	72.8	68.9	73.0	73.8	72.3	92.9	62.0	64.6	69.0	63.1	42.2	69.3
Seed-Coder-8B-Instruct	8B	**85.2**	**72.7**	**77.0**	**74.2**	**72.8**	**78.8**	**74.7**	95.5	**73.4**	**72.5**	**78.0**	**70.3**	54.2	**75.3**
						13B+ Models									
StarCoder2-15B-Instruct	15B	78.0	25.1	25.9	21.7	20.7	59.8	53.5	90.4	46.7	31.9	56.1	43.2	42.0	45.8
Codestral-22B	22B	78.2	73.6	77.3	70.1	71.7	68.5	74.9	97.1	71.0	66.6	74.2	64.4	50.1	72.1
CodeLlama-70B-Instruct	70B	77.8	66.6	68.6	69.2	47.8	62.5	70.5	77.7	57.2	51.1	67.0	51.3	48.7	62.8
DeepSeek-Coder-33B-Instruct	33B	80.4	71.8	76.8	69.9	72.4	69.8	75.1	96.4	70.1	66.6	75.1	64.6	54.3	72.6
DeepSeek-Coder-V2-Lite-Instruct	2.4B/16B	82.8	73.3	75.3	72.4	72.4	73.1	75.1	95.1	69.9	61.6	74.5	63.5	55.0	72.6
DeepSeek-Coder-V2-Instruct	21B/236B	89.4	78.2	77.6	72.6	74.8	**80.5**	75.8	89.1	74.5	70.7	80.2	67.9	59.0	76.2
Qwen2.5-Coder-14B-Instruct	14B	86.2	77.5	84.8	**80.1**	77.6	77.7	79.7	**97.1**	75.3	**76.2**	79.3	**73.1**	**67.2**	79.4
Qwen2.5-Coder-32B-Instruct	32B	**90.2**	**80.4**	**86.3**	73.5	**78.3**	79.3	**87.6**	96.4	**75.6**	74.7	**83.4**	63.3	66.7	**79.7**

Table 8. Performance of instruct models on MBXP.

tasks, the recently proposed benchmark BigCodeBench [Zhuo et al., 2025] challenges LLMs to invoke multiple function calls as tools from 139 libraries across 7 domains, encompassing 1,140 Python tasks with rich context. These high-quality programming tasks demand compositional reasoning and precise comprehension of complex instructions. Each task includes an average of 5.6 test cases, achieving 99% branch coverage on average, thus ensuring rigorous and comprehensive evaluation of LLMs. We evaluated various instruct models on both the full set and hard set of BigCodeBench-Instruct, which is designed to showcase the ability of LLMs to complete coding tasks based on natural language instructions.

As shown in Table 7, our model Seed-Coder-8B-Instruct significantly surpasses other instruct models of comparable size, achieving superior pass@1 scores on both the full set (53.3%) and the hard set (26.4%). Even when compared to models with over 13B parameters, our 8B model outperforms some much larger models, demonstrating the strong code generation capabilities of Seed-Coder-8B-Instruct on challenging and practical tasks.

LiveCodeBench. Static benchmarks for code generation may be subject to potential contamination or overfitting, leading to a skewed or misleading picture of LLMs' actual coding capabilities. To mitigate these issues, LiveCodeBench [Jain et al., 2025] introduces live updates by continuously collecting new problems from coding contests across prominent competitive programming platforms – LeetCode, AtCoder, and CodeForces – and tagging each problem with a release date. Using LiveCodeBench, we can evaluate LLMs on problems selected within a specified, up-to-date time window, effectively preventing contamination and ensuring fair comparisons. We set the LiveCodeBench time window to "2410 – 2502" as it was the most recent available during our evaluation.

Table 7 shows the results of various instruct models on LiveCodeBench within the same specified time window. Our model Seed-Coder-8B-Instruct achieves a standout 24.7% pass@1 score, demonstrating its exceptional performance among the ~8B models. Surprisingly, despite its smaller model size, Seed-Coder-8B-Instruct even surpasses Qwen2.5-Coder-14B-Instruct and some other larger models on LiveCodeBench, highlighting the superior capability of our model in the field of competitive programming.

Model	Size	NCB (zh)			NCB (en)			Total
		Python	*Java*	*Total*	*Python*	*Java*	*Total*	
~8B Models								
CodeLlama-7B-Instruct	7B	18.6	8.6	13.6	17.1	14.3	15.7	14.6
DeepSeek-Coder-6.7B-Instruct	6.7B	38.6	31.4	35.0	32.9	32.9	32.9	33.9
CodeQwen1.5-7B-Chat	7B	30.0	28.6	29.3	30.0	27.1	28.6	25.7
Yi-Coder-9B-Chat	9B	41.4	**45.7**	43.6	38.6	44.3	41.5	42.5
Llama-3.1-8B-Instruct	8B	27.1	24.3	25.7	22.9	22.9	22.9	24.3
OpenCoder-8B-Instruct	8B	40.0	30.0	35.0	35.7	24.3	30.0	32.5
Qwen2.5-Coder-7B-Instruct	7B	34.3	37.1	35.7	34.3	35.7	35.0	35.4
Qwen3-8B	8B	37.1	32.9	35.0	34.3	38.6	36.5	35.7
Seed-Coder-8B-Instruct	8B	**55.7**	45.7	**50.7**	**50.0**	47.1	**48.6**	**49.6**
13B+ Models								
StarCoder2-15B-Instruct	15B	44.3	30.0	37.2	38.6	42.9	40.8	39.0
Codestral-22B	22B	40.0	44.3	42.2	41.4	**45.7**	43.6	42.9
CodeLlama-70B-Instruct	70B	35.1	32.1	33.6	32.8	30.5	31.7	32.6
DeepSeek-Coder-33B-Instruct	33B	44.3	38.9	41.6	**44.3**	44.3	**44.3**	43.0
DeepSeek-Coder-V2-Lite-Instruct	2.4B/16B	41.4	47.1	44.3	41.4	37.1	39.3	41.8
Qwen2.5-Coder-14B-Instruct	14B	**48.6**	**48.6**	**48.6**	42.9	**45.7**	**44.3**	**46.4**

Table 9. Performance of instruct models on NaturalCodeBench.

MBXP. The original MBPP benchmark only contains Python tasks, while MBXP [Athiwaratkun et al., 2023] transforms the problems and unit tests in MBPP to 10+ programming languages. To thoroughly assess the multilingual code generation capabilities of LLMs, we evaluated various instruct models on MBXP. Table 8 presents a comprehensive performance overview of MBXP across 13 widely used programming languages. Seed-Coder-8B-Instruct shows prominent results, achieving the highest scores in most languages among the ~8B models. With an average score of 75.3%, our model even outperforms or matches the performance of some much larger models.

NaturalCodeBench. Traditional code generation benchmarks, such as HumanEval and MBPP, are often limited to well-defined coding problems in algorithmic and basic programming, which may fall short in sufficiently capturing the wide range of needs and complexity in practical software engineering problems. To bridge this gap, NaturalCodeBench [Zhang et al., 2024] is designed to mirror the complexity and variety of scenarios in real-world coding tasks. NaturalCodeBench (NCB) meticulously curates a collection of 402 high-quality problems derived from authentic user queries on popular online coding platforms. These problems, available in Python and Java, span six domains: front-end development, algorithms, data science, artificial intelligence, software engineering, and system administration. Beyond basic data structures such as lists and numbers, the test inputs for NCB problems incorporate diverse file types and complex structures, providing a greater challenge for code LLMs. We evaluate various instruct models on NCB to investigate their abilities to solve real-world coding problems.

We evaluated the performance of various instruct models on both the Chinese (zh) and the English (en) versions of NaturalCodeBench, with the results summarized in Table 9. As shown in the table, Seed-Coder-8B-Instruct consistently exhibits dominant performance in the Python split, outperforming not only all the other listed ~8B models but also larger models with 13B+ parameters,

Model	Size	BP	AP	SE	DA	MA	DW	ML	SC	DB	MM	OS	Others	Overall
~8B Models														
CodeLlama-7B-Instruct	7B	21.4	21.7	30.5	34.3	20.3	40.5	8.8	11.8	34.9	15.0	50.0	29.3	27.1
DeepSeek-Coder-6.7B-Instruct	6.7B	34.2	43.4	38.5	58.1	38.1	43.9	33.8	23.9	46.0	38.3	60.3	44.2	41.9
CodeQwen1.5-7B-Chat	7B	36.7	44.9	**46.0**	51.8	29.7	40.8	26.3	24.3	42.1	41.7	48.5	44.7	40.5
Yi-Coder-9B-Chat	9B	39.1	46.0	39.5	65.0	46.5	**49.7**	42.5	34.9	48.4	41.7	58.8	49.5	47.1
Llama-3.1-8B-Instruct	8B	22.8	39.1	36.0	52.3	43.0	35.7	27.5	22.4	45.2	38.3	45.6	37.8	36.8
OpenCoder-8B-Instruct	8B	39.5	49.1	38.0	55.6	36.0	45.9	27.5	26.5	47.6	46.7	45.6	45.7	43.6
Qwen2.5-Coder-7B-Instruct	7B	38.6	53.2	39.0	63.2	49.7	44.6	37.5	33.5	46.8	55.0	63.2	54.3	48.0
Qwen3-8B	8B	41.6	47.4	38.0	64.0	64.0	45.2	37.5	28.7	46.8	48.3	60.3	48.9	47.7
Seed-Coder-8B-Instruct	8B	**52.8**	**64.1**	37.0	**70.6**	52.8	**49.7**	**51.3**	**44.1**	**56.3**	58.3	**67.6**	**58.5**	**55.8**
13B+ Models														
StarCoder2-15B-Instruct	15B	38.4	42.2	29.0	59.9	37.1	41.0	42.5	28.7	54.8	33.3	42.7	45.7	41.8
Codestral-22B	22B	39.3	46.8	41.5	55.8	42.0	43.4	32.5	30.1	49.2	43.3	45.6	54.8	44.3
CodeLlama-70B-Instruct	70B	31.4	33.3	36.0	47.5	34.6	41.3	25.0	36.8	43.7	28.3	36.8	39.4	37.2
DeepSeek-Coder-33B-Instruct	33B	38.4	50.6	35.5	66.0	50.0	**49.5**	43.8	39.7	49.2	53.3	54.4	48.4	48.6
DeepSeek-Coder-V2-Lite-Instruct	2.4B/16B	45.8	57.2	38.5	56.9	52.8	44.6	42.5	33.8	52.4	33.3	50.0	51.6	48.7
DeepSeek-Coder-V2-Instruct	21B/236B	52.8	**63.6**	**43.0**	71.6	**75.9**	47.5	46.3	**52.9**	54.0	51.7	63.2	59.6	**58.1**
Qwen2.5-Coder-14B-Instruct	14B	**53.3**	58.5	41.0	69.5	69.2	46.3	51.3	43.0	49.2	60.0	**69.1**	57.5	55.3
Qwen2.5-Coder-32B-Instruct	32B	51.9	60.9	**43.0**	**73.1**	69.9	47.1	**55.0**	44.9	56.4	**61.7**	61.8	**60.6**	56.9

Table 10. Performance of instruct models on FullStack Bench, which covers major real-world code development domains including Basic Programming (BP), Advanced Programming (AP), Software Engineering (SE), Data Analysis (DA), Mathematics (MA), Desktop and Web Development (DW), Machine Learning (ML), Scientific Computing (SC), Database (DB), Multimedia (MM), Operating System (OS), and Others.

Model	Size	Bash	C++	C#	D	Go	HTML	Java	JS	PHP	Python	R	Ruby	Rust	Scala	SQL	TS	Overall
~8B Models																		
CodeLlama-7B-Instruct	7B	76.7	13.1	43.9	12.0	23.5	54.4	33.0	37.2	27.8	22.0	17.5	21.1	20.0	25.0	41.3	50.0	27.1
DeepSeek-Coder-6.7B-Instruct	6.7B	76.7	23.4	56.1	13.0	52.9	65.0	41.5	65.4	50.0	39.3	31.3	47.4	43.3	28.6	57.5	60.5	41.9
CodeQwen1.5-7B-Chat	7B	73.3	30.4	41.9	15.2	48.5	57.5	**44.8**	60.3	44.4	37.4	15.0	52.6	58.3	46.4	52.5	60.5	40.5
Yi-Coder-9B-Chat	9B	76.7	25.2	**75.3**	13.0	41.2	67.5	42.6	74.4	55.6	45.6	27.5	63.2	51.7	46.4	56.3	65.8	47.1
Llama-3.1-8B-Instruct	8B	73.3	17.3	49.0	16.3	44.1	58.8	34.1	60.3	27.8	36.8	13.8	26.3	35.0	10.7	57.5	47.4	36.8
OpenCoder-8B-Instruct	8B	66.7	29.9	61.6	30.4	48.5	61.3	41.3	70.5	58.3	40.1	25.0	39.5	50.0	51.8	61.3	60.5	43.6
Qwen2.5-Coder-7B-Instruct	7B	**93.3**	25.2	75.3	13.0	54.4	65.6	43.0	71.3	56.1	43.9	27.5	56.7	35.7		60.0	68.4	48.0
Qwen3-8B	8B	86.7	30.4	63.1	26.1	52.9	61.9	41.5	69.2	47.2	47.4	28.8	60.5	53.3	39.3	57.5	68.4	47.7
Seed-Coder-8B-Instruct	8B	80.0	**46.3**	**71.2**	**32.6**	**60.3**	67.5	43.7	**83.3**	47.2	**55.5**	41.3	**73.7**	**66.7**	**60.7**	71.3	**76.3**	**55.8**
13B+ Models																		
StarCoder2-15B-Instruct	15B	56.7	21.0	60.6	29.4	47.1	49.4	31.4	70.5	44.4	42.4	28.8	**71.1**	35.0	32.1	63.8	52.6	41.8
Codestral-22B	22B	60.0	22.0	58.1	6.5	55.9	63.1	41.5	69.2	55.6	42.8	26.3	60.5	58.3	60.7	60.0	60.5	44.3
CodeLlama-70B-Instruct	70B	43.3	16.8	62.1	9.8	35.3	58.1	35.8	53.8	47.2	34.9	18.8	52.6	48.3	26.8	55.0	55.3	37.2
DeepSeek-Coder-33B-Instruct	33B	60.0	26.6	68.2	19.6	57.4	**71.3**	**44.1**	66.7	50.0	48.1	32.5	60.5	50.0	46.4	60.0	57.9	48.6
DeepSeek-Coder-V2-Lite-Instruct	2.4B/16B	50.0	36.5	52.5	27.2	61.8	64.4	41.3	73.1	58.3	49.4	38.8	55.3	53.3	42.9	57.5	60.5	48.7
DeepSeek-Coder-V2-Instruct	21B/236B	83.3	**43.5**	72.7	28.3	**66.2**	69.4	42.4	**80.8**	61.1	**60.4**	41.3	65.8	68.3	**69.6**	66.3	**68.4**	**58.1**
Qwen2.5-Coder-14B-Instruct	14B	**93.3**	39.3	65.7	41.3	63.2	66.3	43.5	**80.8**	**77.8**	56.2	32.5	**71.1**	73.3	42.9	62.5	**68.4**	55.3
Qwen2.5-Coder-32B-Instruct	32B	83.3	36.9	**76.8**	**46.7**	54.4	**71.3**	40.4	79.5	58.3	59.1	35.0	63.2	**76.7**	50.0	**70.0**	57.9	56.9

Table 11. Performance of instruct models on FullStack Bench across programming languages.

such as Qwen2.5-Coder-14B-Instruct. Similarly, for the Java split, Seed-Coder-8B-Instruct achieves remarkable results compared to existing open-source code LLMs. Notably, Seed-Coder-8B-Instruct attains an overall score of 49.6% with 8B parameters, exceeding the 46.4% achieved by Qwen2.5-Coder-14B-Instruct with 14B parameters, which highlights the strong capability of our model in solving challenging real-world coding problems.

FullStack Bench. The limitations in conventional code generation benchmarks (e.g., HumanEval and MBPP) have been further revealed by systematic, tag-based analysis conducted in the recently proposed FullStack Bench [Liu et al., 2024b]. This work samples 500K questions from Stack

Model	Size	Input-CoT	Output-CoT
~8B Models			
CodeLlama-7B-Instruct	7B	36.1	36.2
DeepSeek-Coder-6.7B-Instruct	6.7B	42.6	45.1
CodeQwen1.5-7B-Chat	7B	44.0	38.8
Yi-Coder-9B-Chat	9B	47.5	55.6
Llama-3.1-8B-Instruct	8B	35.6	37.8
OpenCoder-8B-Instruct	8B	39.9	43.0
Qwen2.5-Coder-7B-Instruct	7B	65.8	65.9
Qwen3-8B	8B	**73.8**	**76.9**
Seed-Coder-8B-Instruct	8B	63.3	67.1
13B+ Models			
StarCoder2-15B-Instruct	15B	45.5	60.9
Codestral-22B	22B	61.3	63.5
CodeLlama-70B-Instruct	70B	56.5	57.8
DeepSeek-Coder-33B-Instruct	33B	47.3	50.6
DeepSeek-Coder-V2-Lite-Instruct	2.4B/16B	53.0	52.9
Qwen2.5-Coder-14B-Instruct	14B	**69.5**	**79.5**

Table 12. Performance of instruct models on CRUXEval.

Overflow[3] – a widely used software development community – and annotates each question's application domain using LLMs. Subsequently, eleven mainstream code development domains are identified from these annotated questions, including Software Engineering (SE), Data Analysis (DA), Machine Learning (ML), Database (DB), etc., covering 88.1% of the questions in Stack Overflow. Through this structured tagging framework, conventional benchmarks such as HumanEval and MBPP are demonstrated to narrowly focus on a very limited set of domains, exhibiting inadequate coverage and low diversity across realistic coding scenarios.

To address the limitations of conventional code generation benchmarks, FullStack Bench is designed to comprehensively evaluate the coding capabilities of LLMs across diverse, real-world scenarios. Employing a meticulous annotation procedure complemented by rigorous cross-validation quality control, FullStack Bench comprises 3,374 human-annotated coding problems, covering 11 distinct code development domains and 16 programming languages.

Table 10 and 11 present the results of various instruct models on FullStack Bench across code development domains and programming languages, respectively. As shown in both tables, Seed-Coder-8B-Instruct achieves a remarkable 55.8% pass@1 score, significantly outperforming other ~8B models and even surpassing some state-of-the-art open-source models of larger scale, such as Qwen2.5-Coder-14B-Instruct. On the diverse set of code development domains and programming languages in FullStack Bench, Seed-Coder-8B-Instruct consistently achieves top-in-class performance, highlighting its exceptional generalization ability and robustness across various coding scenarios.

---

[3]https://stackoverflow.com/questions

Model	Size	Aider *tries=2*	CanItEdit *pass@1*
~8B Models			
CodeLlama-7B-Instruct	7B	1.5	25.7
DeepSeek-Coder-6.7B-Instruct	6.7B	44.4	36.9
CodeQwen1.5-7B-Chat	7B	38.3	34.8
Yi-Coder-9B-Chat	9B	54.1	50.5
Llama-3.1-8B-Instruct	8B	33.1	39.5
OpenCoder-8B-Instruct	8B	30.8	39.0
Qwen2.5-Coder-7B-Instruct	7B	**57.9**	49.5
Qwen3-8B	8B	55.6	45.7
Seed-Coder-8B-Instruct	8B	57.1	**50.5**
13B+ Models			
StarCoder2-15B-Instruct	15B	38.2	31.4
Codestral-22B	22B	51.1	52.4
CodeLlama-70B-Instruct	70B	15.0	40.5
DeepSeek-Coder-33B-Instruct	33B	54.5	46.2
DeepSeek-Coder-V2-Lite-Instruct	2.4B/16B	52.6	45.2
Qwen2.5-Coder-14B-Instruct	14B	**69.2**	**52.9**

Table 13. Performance of instruct models on Aider ("whole" format) and CanItEdit.

5.2.2. *Code Reasoning*

CRUXEval. As introduced in Sec 5.1.3, CRUXEval [Gu et al., 2024] is designed to measure LLM's code reasoning capability by requiring the model to predict output from a given input (CRUXEval-O), or predict the input from a given output (CRUXEval-I). We further evaluated instruct models using the CRUXEval benchmark, with the results presented in Table 12. As shown in the table, Seed-Coder-8B-Instruct delivers competitive performance in both the Input-CoT and the Output-CoT settings within the group of approximately 8B-parameter models, though lagging behind the recently released Qwen3-8B on this benchmark, indicating that the great potential of smaller-scale LLMs is yet to be explored.

5.2.3. *Code Editing*

Aider. We employed Aider's code editing benchmark[4] to assess the code editing capability of LLMs. This benchmark is based on a set of 133 coding exercises from Exercism[5], testing the model by asking it to edit existing code and format the modifications so that all its changes to the source file can be successfully applied without human intervention. The comprehensive design of Aider's coding benchmark captures not only the coding proficiency of LLMs but also their consistency in generating code modifications aligned precisely with prompt specifications.

Aider uses different edit formats to collect code edits from different LLMs. For a fair comparison, we used the "whole" format with a default *tries=2* setting for all evaluations. Table 13 outlines

---

[4]https://aider.chat/docs/leaderboards/edit.html
[5]https://github.com/exercism/python

Model	Size	Debug	Translate	Switch	Polish
~8B Models					
CodeLlama-7B-Instruct	7B	15.5	28.9	1.7	1.2
DeepSeek-Coder-6.7B-Instruct	6.7B	24.5	33.8	13.7	1.5
CodeQwen1.5-7B-Chat	7B	21.1	36.8	11.6	1.1
Yi-Coder-9B-Chat	9B	27.0	36.1	15.6	1.2
Llama-3.1-8B-Instruct	8B	20.8	25.0	2.5	1.5
OpenCoder-8B-Instruct	8B	27.2	37.9	13.5	0.7
Qwen2.5-Coder-7B-Instruct	7B	26.2	44.5	14.1	1.3
Qwen3-8B	8B	27.9	44.9	11.2	1.5
Seed-Coder-8B-Instruct	8B	**30.7**	**56.5**	**17.7**	**1.9**
13B+ Models					
StarCoder2-15B-Instruct	15B	19.0	42.0	8.1	1.2
Codestral-22B	22B	27.6	39.5	14.1	1.0
CodeLlama-70B-Instruct	70B	3.5	21.0	5.5	0.3
DeepSeek-Coder-33B-Instruct	33B	27.7	41.4	**17.2**	1.1
DeepSeek-Coder-V2-Lite-Instruct	2.4B/16B	28.0	34.1	15.0	1.1
Qwen2.5-Coder-14B-Instruct	14B	**29.7**	**50.8**	16.1	**1.7**

Table 14. Performance of instruct models on CodeEditorBench.

the performance of various instruct models on Aider's code editing benchmark. We utilized scores from the official Aider leaderboard whenever available and adopted the reported scores for additional models provided by Qwen2.5-Coder [Hui et al., 2024]. In cases where discrepancies arise between these two sources (e.g., Qwen2.5-Coder-7B-Instruct self-reported 68.4%, whereas the official leaderboard lists 57.9%), we defaulted to the official leaderboard scores. As shown in the table, Seed-Coder-8B-Instruct achieves top-tier performance, comparable to Qwen2.5-Coder-7B-Instruct, and clearly outperforms other models of similar size, including Qwen3-8B, which was just released very recently.

CanItEdit. To assess LLM's proficiency in handling diverse code editing scenarios, we incorporated the CanItEdit benchmark [Cassano et al., 2024] in our code editing evaluations. CanItEdit serves as a benchmark for evaluating the performance of LLMs in instructional code editing, which comprises 105 hand-crafted instructional code editing problems, featuring both descriptive and lazy instructions. As shown in Table 13, Seed-Coder-8B-Instruct also takes the lead among all the ~8B models, demonstrating the outstanding code editing capability of our model.

CodeEditorBench. A recent code editing benchmark CodeEditorBench [Guo et al., 2025] further categorizes the code editing task into four key scenarios: code debugging, code translation, code requirement switching, and code polishing. CodeEditorBench is designed to systematically assess the code editing capability of LLMs across these key scenarios. We employed CodeEditorBench to evaluate various instruct models, with the results presented in Table 14. As shown in the table, Seed-Coder-8B-Instruct achieves top-notch performance across the four key code editing scenarios, even surpassing larger state-of-the-art open-source models such as Qwen2.5-Coder-14B-Instruct. The results obtained on these code editing benchmarks establish our model as the best-performing open-source code editing model at ~8B scale.

| Model | Size | SWE-bench Verified | | Multi-SWE-bench mini |
		Agentless	OpenHands	Agentless
~8B Models				
Yi-Coder-9B-Chat	9B	0.0	1.6	0.0
Llama-3.1-8B-Instruct	8B	1.0	1.2	0.5
Qwen2.5-Coder-7B-Instruct	7B	4.2	1.0	0.5
Qwen3-8B	8B	14.6	3.4	2.3
Qwen3-8B-Thinking	8B	12.5	8.6	2.5
Seed-Coder-8B-Instruct	8B	**19.2**	**11.2**	**4.0**
13B+ Models				
StarCoder2-15B-Instruct	15B	0.0	0.0	0.0
CodeLlama-34B-Instruct	34B	0.2	0.8	0.5
DeepSeek-Coder-V2-Lite-Instruct	2.4B/16B	4.4	1.0	0.5
Qwen2.5-Coder-14B-Instruct	14B	19.4	1.4	3.8
Qwen2.5-Coder-32B-Instruct	32B	**30.2**	5.6	**4.5**
QwQ-32B	32B	18.4	**15.8**	**4.5**

Table 15. Performance of instruct models on SWE-bench Verified and Multi-SWE-bench mini.

5.2.4. Software Engineering

We evaluated Seed-Coder and competitor models on two benchmarks to assess their effectiveness in realistic software engineering scenarios:

- **SWE-bench Verified** [Jimenez et al., 2024] is a benchmark designed to evaluate the ability of LLMs to solve real-world GitHub issues. It consists of 500 Python instances, each manually verified to ensure the accuracy of both the issue description and the corresponding patch.

- **Multi-SWE-bench** [Zan et al., 2025] is a recently proposed new benchmark, aiming to evaluate the ability of LLMs to resolve issues across multiple programming languages. It comprises 1,632 manually verified instances across 8 languages: Python, Java, TypeScript, JavaScript, Go, Rust, C, and C++. We evaluated on the *mini* version – a balanced subset of Multi-SWE-bench consisting of 400 instances, with 50 instances per language. Multi-SWE-bench serves as a practical benchmark for testing the multilingual capabilities of LLMs in real-world software engineering tasks.

Two representative methods of agent scaffolding were used in our evaluation:

- **Agentless** [Xia et al., 2024] is a method that follows a fixed, manually designed workflow. It decomposes the issue resolving task into pre-defined standard operating procedure consisting of fault localization, code repair, and patch validation.
- **OpenHands** [Wang et al., 2025] is a fully autonomous agent platform that has no requirements of any fixed workflow. It relies entirely on LLM's own planning and reasoning capability, using minimal prompting and external tools to plan, act, and iterate in solving issues.

As shown in Table 15, Seed-Coder-8B-Instruct achieves the strongest performance among all ~8B models on SWE-bench Verified and Multi-SWE-bench mini, with 19.2% and 4.0% resolved rate under Agentless. Despite its smaller size, Seed-Coder-8B-Instruct matches or exceeds the performance of much larger models such as QwQ-32B (18.4%) and Qwen2.5-Coder-14B-Instruct (19.4%), highlighting its parameter efficiency and strong code generation capabilities.

We further noticed a consistent pattern: Agentless outperforms OpenHands in most cases. This gap is primarily due to the limited capacity of smaller models, which typically rely on pre-defined workflows (Agentless) to perform reliably and often fail when required to operate autonomously (OpenHands). However, as the model capability continuously improves, agent-based methods like OpenHands are expected to surpass fixed workflows, as reasoning and control flow become increasingly internalized within the model's parameters, enabling more dynamic and adaptive solutions.

Notably, Seed-Coder-8B-Instruct breaks the norm. It achieves 11.2% resolved rate under OpenHands, significantly outperforming all the other ~8B baselines, further demonstrating its ability to operate effectively without pre-defined workflow. On Multi-SWE-bench mini, Seed-Coder-8B-Instruct also achieves the highest score among all models of comparable scale. This further demonstrates the model's ability to generalize across languages in software engineering scenarios. We attribute this effectiveness to two key factors: (1) strong instruction-following capabilities, reinforced by applying LLM filter to ensure consistent formatting across training data; (2) enhanced software engineering skills, achieved by incorporating commit data during training to support fault localization and code repair. Overall, Seed-Coder delivered strong code generation performance on software engineering tasks, demonstrating the potential of small models to tackle complex, real-world scenarios.

5.3. Evaluation of Reasoning Models

To evaluate the effectiveness of our Reasoning model, we conducted experiments on several mainstream and challenging tasks, including LiveCodeBench (2410 – 2502), the International Olympiad in Informatics[6] (IOI), and Codeforces[7]. For IOI, we adopted the evaluation protocol from Open R1 [Hugging Face, 2025], performing tests on 41 subtasks from the IOI'2024 competition and reporting results based on full submissions. For Codeforces, we followed the scoring methodology and used contest problems consistent with the standardized competition-level code generation benchmark CodeElo [Quan et al., 2025], implemented within our self-developed submission system, enabling direct comparisons.

As shown in Table 16, our distilled model DeepSeek-R1-Distill-Seed-Coder-8B, trained solely with warmup data, achieves an overall pass@1 score of 39.0% on LiveCodeBench, surpassing similarly distilled models such as DeepSeek-R1-Distill-Qwen-7B and OlympicCoder-7B. Although we observed that scaling the distillation dataset could lead to further performance improvements, we chose not to further expand the dataset size in order to leave room for reinforcement learning (RL). Extensive distillation would significantly alter the distribution of the base model and might obscure the model's true upper bound capabilities.

After RL training, our model Seed-Coder-8B-Reasoning markedly improves the overall pass@1 score on LiveCodeBench by 14.6 points. With an overall pass@1 score of 53.6%, our model surpasses larger models such as DeepSeek-R1-Distill-Qwen-14B, and even slightly outperforms Claude-3.7-Sonnet-Thinking. Notably, during the RL training, our model exhibits substantial pass@1 improvements for medium and hard questions, increasing by 22.8 and 14.0 points, respectively. Furthermore, the scores on hard questions remain stable across different evaluation periods, with a distribution closely matching that of Claude-3.7-Sonnet-Thinking.

---

[6]https://github.com/huggingface/ioi
[7]https://codeforces.com

Model	Hard			Medium			Easy			Overall
	4-mon	*3-mon*	*2-mon*	*4-mon*	*3-mon*	*2-mon*	*4-mon*	*3-mon*	*2-mon*	
~8B Models										
DeepSeek-R1-Distill-Qwen-7B	11.3	10.7	9.6	39.6	37.2	37.1	76.2	77.1	67.1	36.5
DeepSeek-R1-Distill-Seed-Coder-8B	13.6	13.9	13.4	39.6	38.7	39.3	79.8	80.2	73.2	39.0
OlympicCoder-7B	12.7	11.8	12.5	40.8	39.0	38.7	78.0	77.1	67.8	37.9
Qwen3-8B-Thinking	27.5	23.5	19.7	65.7	**59.7**	**58.5**	**98.0**	**98.1**	97.3	**57.4**
Seed-Coder-8B-Reasoning	**27.6**	**28.0**	**31.0**	**65.8**	59.2	57.5	87.8	88.0	80.1	53.6
13B+ Models										
DeepSeek-R1-Distill-Qwen-14B	21.3	20.5	16.1	58.1	53.4	51.4	93.3	94.2	93.7	51.9
Claude-3.7-Sonnet-Thinking	27.3	30.8	**31.0**	54.5	55.1	51.4	96.2	**100.0**	**100.0**	53.3
o3-mini-low	**30.3**	**32.3**	28.6	**69.6**	**61.2**	**54.1**	**98.7**	**100.0**	**100.0**	**59.4**

Table 16. Performance (pass@1) of reasoning models on LiveCodeBench evaluated over different time windows: 4-mon (2410 – 2502), 3-mon (2411 – 2502), and 2-mon (2412 – 2502).
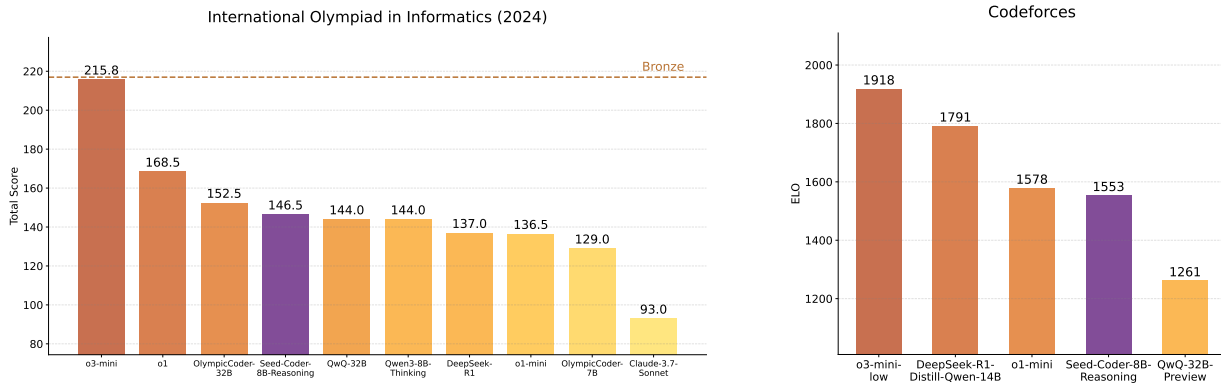
Figure 11. Performance of reasoning models on IOI'2024 (left) and Codeforces (right).

The evaluation results of IOI and Codeforces are shown in Figure 11. On the IOI benchmark, our model achieves higher scores than QwQ-32B and the 671B-sized DeepSeek-R1, also outperforming similarly-sized thinking models. In the Codeforces evaluations, our model scores 1553 points, closely aligning with o1-mini and significantly surpassing the powerful QwQ-32B-Preview model. These results are largely consistent with the IOI rankings. Additionally, we noticed that more than 10% of the samples were truncated at the 64K sequence length, suggesting a potentially higher performance ceiling of our model.

However, we also observed notable gaps between the models' scores and the IOI bronze medal threshold, indicating that substantial capability improvements are still required, particularly for solving difficult subtasks where models of similar sizes typically fail to generate correct solutions. Additionally, our model's accuracy on Codeforces problems rated above 2000 points remains very low, and its performance around the 88th percentile still exhibits a significant gap compared to top-tier competitors.

6. Conclusion, Limitation, and Future Work

In this report, we introduce Seed-Coder, a family of lightweight yet powerful open-source code LLMs achieving state-of-the-art performance across diverse coding benchmarks. We demonstrate that, with minimal human effort, LLMs can effectively curate training data by themselves to significantly enhance code intelligence and reasoning capabilities.

Seed-Coder represents our initial step towards contributing to the open-source LLM community. As Seed-Coder primarily focuses on coding tasks and excludes general web data during training, its general natural language understanding and ability to handle broader tasks remain limited. Furthermore, the substantial token volume disparity compared to other models (e.g., Qwen3 is pretrained on 36 trillion tokens, while Seed-Coder is only pretrained on 6 trillion tokens) stems from insufficient general knowledge and mathematical data, imposing inherent comprehension constraints. We view this release as the foundation of a growing model family, with future iterations on further improving coding capabilities spanning a range of model sizes.

References

01.AI. Meet Yi-Coder: A small but mighty LLM for code, September 2024. URL https://github.com/01-ai/Yi-Coder.

L. B. Allal, R. Li, D. Kocetkov, C. Mou, C. Akiki, C. M. Ferrandis, N. Muennighoff, M. Mishra, A. Gu, M. Dey, et al. SantaCoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988*, 2023. URL https://arxiv.org/abs/2301.03988.

Anthropic. Claude 3.7 sonnet, February 2025. URL https://www.anthropic.com/claude.

B. Athiwaratkun, S. K. Gouda, Z. Wang, X. Li, Y. Tian, M. Tan, W. U. Ahmad, S. Wang, Q. Sun, M. Shang, S. K. Gonugondla, H. Ding, V. Kumar, N. Fulton, A. Farahani, S. Jain, R. Giaquinto, H. Qian, M. K. Ramanathan, R. Nallapati, B. Ray, P. Bhatia, S. Sengupta, D. Roth, and B. Xiang. Multi-lingual evaluation of code generation models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=Bo7eeXm6An8.

J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, et al. Program synthesis with large language models. *ArXiv preprint*, abs/2108.07732, 2021. URL https://arxiv.org/abs/2108.07732.

M. Bavarian, H. Jun, N. Tezak, J. Schulman, C. McLeavey, J. Tworek, and M. Chen. Efficient training of language models to fill in the middle. *arXiv preprint arXiv:2207.14255*, 2022a. URL https://arxiv.org/abs/2207.14255.

M. Bavarian, H. Jun, N. Tezak, J. Schulman, C. McLeavey, J. Tworek, and M. Chen. Efficient training of language models to fill in the middle. *arXiv preprint arXiv:2207.14255*, 2022b.

M. Brunsfeld. Tree-sitter: An incremental parsing system for programming tools, 2018. URL https://tree-sitter.github.io/tree-sitter/.

ByteDance Seed. Seed1.5-Thinking: Advancing superb reasoning models with reinforcement learning, 2025. URL https://arxiv.org/abs/2504.13914.

F. Cassano, J. Gouwar, D. Nguyen, S. Nguyen, L. Phipps-Costin, D. Pinckney, M.-H. Yee, Y. Zi, C. J. Anderson, M. Q. Feldman, A. Guha, M. Greenberg, and A. Jangda. MultiPL-E: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7):3675–3691, 2023. doi: 10.1109/TSE.2023.3267446. URL https://www.computer.org/csdl/journal/ts/2023/07/10103177/1MpWUtj7Rwk.

F. Cassano, L. Li, A. Sethi, N. Shinn, A. Brennan-Jones, J. Ginesin, E. Berman, G. Chakhnashvili, A. Lozhkov, C. J. Anderson, and A. Guha. Can it edit? evaluating the ability of large language models to follow code editing instructions. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=D06yk3DBas.

M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

J. Dai, J. Lu, Y. Feng, D. Huang, G. Zeng, R. Ruan, M. Cheng, H. Tan, and Z. Guo. MHPP: Exploring the capabilities and limitations of language models beyond basic code generation, 2024. URL https://arxiv.org/abs/2405.11430.

DeepSeek-AI, A. Liu, B. Feng, B. Wang, B. Wang, B. Liu, and C. Z. et al. DeepSeek-V2: A strong, economical, and efficient mixture-of-experts language model, 2024a. URL https://arxiv.org/abs/2405.04434.

DeepSeek-AI, Q. Zhu, D. Guo, Z. Shao, D. Yang, P. Wang, R. Xu, Y. Wu, Y. Li, H. Gao, S. Ma, W. Zeng, X. Bi, Z. Gu, H. Xu, D. Dai, K. Dong, L. Zhang, Y. Piao, Z. Gou, Z. Xie, Z. Hao, B. Wang, J. Song, D. Chen, X. Xie, K. Guan, Y. You, A. Liu, Q. Du, W. Gao, X. Lu, Q. Chen, Y. Wang, C. Deng, J. Li, C. Zhao, C. Ruan, F. Luo, and W. Liang. DeepSeek-Coder-V2: Breaking the barrier of closed-source models in code intelligence, 2024b. URL https://arxiv.org/abs/2406.11931.

DeepSeek-AI, D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi, and X. Z. et al. DeepSeek-R1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL https://arxiv.org/abs/2501.12948.

Y. Ding, Z. Wang, W. U. Ahmad, H. Ding, M. Tan, N. Jain, M. K. Ramanathan, R. Nallapati, P. Bhatia, D. Roth, and B. Xiang. CrossCodeEval: A diverse and multilingual benchmark for cross-file code completion. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/920f2dced7d32ab2ba2f1970bc306af6-Abstract-Datasets_and_Benchmarks.html.

D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, S. Yih, L. Zettlemoyer, and M. Lewis. InCoder: A generative model for code infilling and synthesis. In *The Eleventh*

International Conference on Learning Representations, 2023. URL https://openreview.net/forum?id=hQwb-lbM6EL.

A. Gu, B. Rozière, H. Leather, A. Solar-Lezama, G. Synnaeve, and S. I. Wang. CRUXEval: A benchmark for code reasoning, understanding and execution, 2024. URL https://arxiv.org/abs/2401.03065.

D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, F. Luo, Y. Xiong, and W. Liang. DeepSeek-Coder: When the large language model meets programming – the rise of code intelligence, 2024. URL https://arxiv.org/abs/2401.14196.

J. Guo, Z. Li, X. Liu, K. Ma, T. Zheng, Z. Yu, D. Pan, Y. LI, R. Liu, Y. Wang, S. Guo, X. Qu, X. Yue, G. Zhang, W. Chen, and J. Fu. CodeEditorBench: Evaluating code editing capability of LLMs. In *ICLR 2025 Third Workshop on Deep Learning for Code*, 2025. URL https://openreview.net/forum?id=6yTgoh0J0X.

C.-Y. Hsieh, Y.-S. Chuang, C.-L. Li, Z. Wang, L. T. Le, A. Kumar, J. Glass, A. Ratner, C.-Y. Lee, R. Krishna, and T. Pfister. Found in the middle: Calibrating positional attention bias improves long context utilization, 2024. URL https://arxiv.org/abs/2406.16008.

S. Huang, T. Cheng, J. K. Liu, J. Hao, L. Song, Y. Xu, J. Yang, J. Liu, C. Zhang, L. Chai, R. Yuan, Z. Zhang, J. Fu, Q. Liu, G. Zhang, Z. Wang, Y. Qi, Y. Xu, and W. Chu. OpenCoder: The open cookbook for top-tier code large language models, 2025. URL https://arxiv.org/abs/2411.04905.

Hugging Face. Open R1: A fully open reproduction of deepseek-r1, January 2025. URL https://github.com/huggingface/open-r1.

B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Lu, K. Dang, Y. Fan, Y. Zhang, A. Yang, R. Men, F. Huang, B. Zheng, Y. Miao, S. Quan, Y. Feng, X. Ren, X. Ren, J. Zhou, and J. Lin. Qwen2.5-Coder technical report, 2024. URL https://arxiv.org/abs/2409.12186.

N. Jain, K. Han, A. Gu, W.-D. Li, F. Yan, T. Zhang, S. Wang, A. Solar-Lezama, K. Sen, and I. Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=chfJJYC3iL.

C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

A. Joulin, E. Grave, P. Bojanowski, M. Douze, H. Jégou, and T. Mikolov. FastText.zip: Compressing text classification models, 2016. URL https://arxiv.org/abs/1612.03651.

A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov. Bag of tricks for efficient text classification. In M. Lapata, P. Blunsom, and A. Koller, editors, *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*, pages 427–431, Valencia, Spain, Apr. 2017. Association for Computational Linguistics. URL https://aclanthology.org/E17-2068/.

R. Kiveris, S. Lattanzi, V. Mirrokni, V. Rastogi, and S. Vassilvitskii. Connected components in mapreduce and beyond. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, page 1–13, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450332521. doi: 10.1145/2670979.2670997. URL https://doi.org/10.1145/2670979.2670997.

K. Lee, D. Ippolito, A. Nystrom, C. Zhang, D. Eck, C. Callison-Burch, and N. Carlini. Deduplicating training data makes language models better. In S. Muresan, P. Nakov, and A. Villavicencio, editors, *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8424–8445, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.acl-long.577. URL https://aclanthology.org/2022.acl-long.577/.

R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, et al. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023. URL https://arxiv.org/abs/2305.06161.

Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago, et al. Competition-level code generation with AlphaCode. *Science*, 378(6624): 1092–1097, 2022. URL https://www.science.org/doi/10.1126/science.abq1158.

J. Liu, C. S. Xia, Y. Wang, and L. Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/43e9d647ccd3e4b7b5baab53f0368686-Abstract-Conference.html.

R. Liu, J. Wei, F. Liu, C. Si, Y. Zhang, J. Rao, S. Zheng, D. Peng, D. Yang, D. Zhou, and A. M. Dai. Best practices and lessons learned on synthetic data, 2024a. URL https://arxiv.org/abs/2404.07503.

S. Liu, H. Zhu, J. Liu, S. Xin, A. Li, R. Long, L. Chen, J. Yang, J. Xia, Z. Y. Peng, S. Liu, Z. Zhang, G. Zhang, W. Huang, K. Shen, and L. Xiang. Fullstack bench: Evaluating llms as full stack coders, 2024b. URL https://arxiv.org/abs/2412.00535.

Llama Team. Introducing meta llama 3: The most capable openly available llm to date. https://ai.meta.com/blog/meta-llama-3/, April 2024a.

Llama Team. Introducing llama 3.1: Our most capable models to date. https://ai.meta.com/blog/meta-llama-3-1/, July 2024b.

A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei, T. Liu, M. Tian, D. Kocetkov, A. Zucker, Y. Belkada, Z. Wang, Q. Liu, D. Abulkhanov, I. Paul, Z. Li, W.-D. Li, M. Risdal, J. Li, J. Zhu, T. Y. Zhuo, E. Zheltonozhskii, N. O. O. Dade, W. Yu, L. Krauß, N. Jain, Y. Su, X. He, M. Dey, E. Abati, Y. Chai, N. Muennighoff, X. Tang, M. Oblokulov, C. Akiki, M. Marone, C. Mou, M. Mishra, A. Gu, B. Hui, T. Dao, A. Zebaze, O. Dehaene, N. Patry, C. Xu, J. McAuley, H. Hu, T. Scholak, S. Paquet, J. Robinson, C. J. Anderson, N. Chapados, M. Patwary, N. Tajbakhsh, Y. Jernite, C. M. Ferrandis, L. Zhang, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries. StarCoder 2 and The Stack v2: The next generation, 2024. URL https://arxiv.org/abs/2402.19173.

K. Lu, H. Yuan, Z. Yuan, R. Lin, J. Lin, C. Tan, C. Zhou, and J. Zhou. InsTag: Instruction tagging for analyzing supervised fine-tuning of large language models, 2023. URL https://arxiv.org/abs/2308.07074.

M. Luo, S. Tan, R. Huang, X. Shi, R. Xin, C. Cai, A. Patel, A. Ariyak, Q. Wu, C. Zhang, L. E. Li, R. A. Popa, and I. Stoica. DeepCoder: A fully open-source 14B coder at o3-mini level. https://pretty-radio-b75.notion.site/DeepCoder-A-Fully-Open-Source-14B-Coder-at-O3-min i-Level-1cf81902c14680b3bee5eb349a512a51, 2025. Notion Blog.

Mistral AI. Codestral. https://mistral.ai/news/codestral/, May 2024.

OpenAI. OpenAI o3, April 2025. URL https://openai.com/index/introducing-o3-and-o4-mini/.

G. Penedo, A. Lozhkov, H. Kydlíček, L. Ben Allal, E. Beeching, A. Piqueres Lajarín, Q. Gallouédec, N. Habib, L. Tunstall, and L. von Werra. CodeForces-CoTs. https://huggingface.co/datasets/op en-r1/codeforces-cots, 2025. Accessed: 2025-04-26.

S. Quan, J. Yang, B. Yu, B. Zheng, D. Liu, A. Yang, X. Ren, B. Gao, Y. Miao, Y. Feng, Z. Wang, J. Yang, Z. Cui, Y. Fan, Y. Zhang, B. Hui, and J. Lin. CodeElo: Benchmarking competition-level code generation of llms with human-comparable elo ratings, 2025. URL https://arxiv.org/abs/2501.01257.

Qwen Team. Code with CodeQwen1.5, April 2024a. URL https://qwenlm.github.io/blog/codeqw en1.5/.

Qwen Team. Introducing Qwen1.5, February 2024b. URL https://qwenlm.github.io/blog/qwen1.5/.

Qwen Team. Qwen2.5: A party of foundation models, September 2024c. URL https://qwenlm.git hub.io/blog/qwen2.5/.

Qwen Team. Qwen3, April 2025. URL https://qwenlm.github.io/blog/qwen3/.

R. Rafailov, A. Sharma, E. Mitchell, C. D. Manning, S. Ermon, and C. Finn. Direct preference optimization: Your language model is secretly a reward model. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=HPuSIXJaa9.

S. Robertson and H. Zaragoza. The probabilistic relevance framework: Bm25 and beyond. *Found. Trends Inf. Retr.*, 3(4):333–389, Apr. 2009. ISSN 1554-0669. doi: 10.1561/1500000019. URL https://doi.org/10.1561/1500000019.

B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve. Code Llama: Open foundation models for code, 2024. URL https://arxiv.org/abs/2308.12950.

Z. Shao, P. Wang, Q. Zhu, R. Xu, J. Song, X. Bi, H. Zhang, M. Zhang, Y. K. Li, Y. Wu, and D. Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.

G. Sheng, C. Zhang, Z. Ye, X. Wu, W. Zhang, R. Zhang, Y. Peng, H. Lin, and C. Wu. HybridFlow: A flexible and efficient RLHF framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 1279–1297. ACM, Mar. 2025. doi: 10.1145/3689031.3696075. URL http://dx.doi.org/10.1145/3689031.3696075.

R. Sutton. The bitter lesson, 2019. URL http://www.incompleteideas.net/IncIdeas/BitterLesson.html.

H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023. URL https://arxiv.org/abs/2307.09288.

X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh, H. H. Tran, F. Li, R. Ma, M. Zheng, B. Qian, Y. Shao, N. Muennighoff, Y. Zhang, B. Hui, J. Lin, R. Brennan, H. Peng, H. Ji, and G. Neubig. OpenHands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=OJd3ayDDoF.

Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang. Magicoder: Empowering code generation with oss-instruct, 2024. URL https://arxiv.org/abs/2312.02120.

C. S. Xia, Y. Deng, S. Dunn, and L. Zhang. Agentless: Demystifying LLM-based software engineering agents, 2024. URL https://arxiv.org/abs/2407.01489.

Q. Yu, Z. Zhang, R. Zhu, Y. Yuan, X. Zuo, Y. Yue, T. Fan, G. Liu, L. Liu, X. Liu, H. Lin, Z. Lin, B. Ma, G. Sheng, Y. Tong, C. Zhang, M. Zhang, W. Zhang, H. Zhu, J. Zhu, J. Chen, J. Chen, C. Wang, H. Yu, W. Dai, Y. Song, X. Wei, H. Zhou, J. Liu, W.-Y. Ma, Y.-Q. Zhang, L. Yan, M. Qiao, Y. Wu, and M. Wang. DAPO: An open-source LLM reinforcement learning system at scale, 2025. URL https://arxiv.org/abs/2503.14476.

Z. Yu, Z. Wang, Y. Fu, H. Shi, K. Shaikh, and Y. C. Lin. Unveiling and harnessing hidden attention sinks: enhancing large language models without training through attention calibration. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024. URL https://dl.acm.org/doi/10.5555/3692070.3694448.

Y. Yue, Z. Chen, R. Lu, A. Zhao, Z. Wang, Y. Yue, S. Song, and G. Huang. Does reinforcement learning really incentivize reasoning capacity in LLMs beyond the base model?, 2025. URL https://arxiv.org/abs/2504.13837.

D. Zan, Z. Huang, W. Liu, H. Chen, L. Zhang, S. Xin, L. Chen, Q. Liu, X. Zhong, A. Li, S. Liu, Y. Xiao, L. Chen, Y. Zhang, J. Su, T. Liu, R. Long, K. Shen, and L. Xiang. Multi-SWE-bench: A multilingual benchmark for issue resolving, 2025. URL https://arxiv.org/abs/2504.02605.

F. Zhang, B. Chen, Y. Zhang, J. Liu, D. Zan, Y. Mao, J.-G. Lou, and W. Chen. RepoCoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570*, 2023. URL https://arxiv.org/abs/2303.12570.

S. Zhang, H. Zhao, X. Liu, Q. Zheng, Z. Qi, X. Gu, X. Zhang, Y. Dong, and J. Tang. NaturalCodeBench: Examining coding performance mismatch on humaneval and natural user prompts, 2024. URL https://arxiv.org/abs/2405.04520.

W. Zhao, X. Ren, J. Hessel, C. Cardie, Y. Choi, and Y. Deng. WildChat: 1m chatgpt interaction logs in the wild, 2024. URL https://arxiv.org/abs/2405.01470.

T. Y. Zhuo, V. M. Chien, J. Chim, H. Hu, W. Yu, R. Widyasari, I. N. B. Yusuf, H. Zhan, J. He, I. Paul, S. Brunner, C. GONG, J. Hoang, A. R. Zebaze, X. Hong, W.-D. Li, J. Kaddour, M. Xu, Z. Zhang, P. Yadav, N. Jain, A. Gu, Z. Cheng, J. Liu, Q. Liu, Z. Wang, D. Lo, B. Hui, N. Muennighoff, D. Fried, X. Du, H. de Vries, and L. V. Werra. BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=YrycTjllL0.

7. Contributions and Acknowledgments

Acknowledgments

A. Appendix

A.1. Programming Languages in GitHub Data

89 programming languages were used in our GitHub Data for pretraining. The full list is as follows:

ANTLR, Ada, Agda, Alloy, AppleScript, Assembly, Augeas, AWK, Batchfile, Bluespec, C, C#, C++, CMake, CSS, Clojure, CoffeeScript, Common Lisp, CUDA, Dart, Dockerfile, Elixir, Elm, Emacs Lisp, Erlang, F#, Fortran, GLSL, Go, Groovy, HTML, Haskell, Idris, Isabelle, JSON, Java, Java Server Pages, JavaScript, Julia, Kotlin, Lean, Literate Agda, Literate CoffeeScript, Literate Haskell, Lua, Makefile, Maple, Markdown, Mathematica, MATLAB, OCaml, PHP, Pascal, Perl, PowerShell, Prolog, Protocol Buffer, Python, R, RMarkdown, Racket, Ruby, Rust, SAS, SPARQL, SQL, Scala, Scheme, Shell, Smalltalk, Solidity, Stan, Standard ML, Stata, Swift, SystemVerilog, Tcl, Tcsh, TeX, Thrift, TypeScript, VHDL, Verilog, Visual Basic, XSLT, YAML, Yacc, Zig, reStructuredText.

A.2. Configuration of Quality Scorer

In this section, we provide the implementation details of the quality scorer described in Section 2.2.1.

A.2.1. Prompt for GitHub Code Quality Scoring

This is the original prompt we used to evaluate the quality of individual code files. It remains consistent throughout the entire pipeline, from collecting ground-truth data to training the quality scorer and applying it across all GitHub data during inference.

```
You are an expert of coding. Please carefully evaluate the quality of the {LANGUAGE} code file below based
on the specific quality criteria essential for its potential use in pretraining a large language model.
Begin your assessment with a brief explanation that addresses the key factors listed below. Following your
explanation, assign a numerical rating to the code file on a scale from 1 to 10, where 1 indicates the
lowest quality and 10 indicates the highest quality. Please adhere strictly to the following format for
your rating: "Rating: [[X]]", where X is your numerical rating. Note that the zero score policy should
be firstly considered in your analysis, and skip the other criteria if the code meets any zero score
conditions.
Criteria for Evaluation:
* Readability:
- Presence of a reasonable amount of comments.
- Inclusion of classes or functions, better with reasonable docstrings that describe the functionality.
- Neat and consistent formatting that adheres to common practice.
- Good naming conventions and well-structured code.
* Modularity:
- Avoidance of overly complicated / very long functions through modularization.
- Clear separation of logic and functionality, using classes and functions.
- Design of each module or component to perform a clear and coherent task.
* Clarity:
- Minimization of excessively repeated code and code blocks, such as repeatedly calling the same function
for many times.
- Avoidance of massive commented-out code blocks.
```

```
- Avoidance of many random printing statements for debugging.
- Clear communication of intentions behind code blocks.
* Reusability:
- Absence of syntax or logical errors.
- Avoidance of embedding lots of hard-coded data directly within the code.
- Provision of complete and meaningful functionality, not overly simplistic.
- Design that facilitates easy reuse of functions or classes in other projects.
* Zero Score Policy:
- If the code is mostly configurations, such as very long json objects with many numbers or strings, rate
0 score.
- If the code is essentially a data file which includes lots of hard-coded data, such as too many lines of
numbers or strings, rate 0 score.
- If the code has little to none effective logic, or is dominated by literals or assignments without any
complexity, rate 0 score.
- If the code is auto-generated, with any comments like "generated by Django", rate 0 score.
After your analysis, provide your explanation for the aspects evaluated. Then, conclude with the rating in
the specified format. For example, if you rate the code quality as 5 out of 10, you should write: "Rating:
[[5]]". {LANGUAGE} code to be assessed: {CONTENT}.
```

The LANGUAGE parameter specifies the primary language of a file, inferred from its extension. The CONTENT parameter contains the file's original content as a string.

A.2.2. *Ground-Truth Quality Scores and Comparison of Oracles*

We collected ground-truth scores from 21 representative programming languages. Table 17 shows the number of files for each language.

Language	Count	Language	Count	Language	Count
Python	26,924	C#	10,112	reStructuredText	9,951
Shell	10,194	SQL	10,110	C++	9,938
Ruby	10,187	JavaScript	10,110	HTML	9,422
Go	10,185	CSS	10,053	R	9,379
TypeScript	10,179	Kotlin	10,051	TeX	9,156
MATLAB	10,165	PHP	10,049	Markdown	8,845
Java	10,161	C	10,020	R Markdown	6,875

Table 17. Language distribution of the quality scorer training set.

Figure 12 compares the ground-truth scores from the responses of three oracles. We used GPT-4 Turbo as the baseline, comparing it with DeepSeek-Coder-33B and DeepSeek-V2-Chat. We observed that despite minor discrepancies in determining whether a file meets the zero score conditions, all models produced consistent scores, both on a per-sample basis and in the overall distribution. In practice, we used DeepSeek-V2-Chat as the oracle for training the quality scorer, balancing efficiency and accuracy.

Distribution of DeepSeek-Coder Scores

GPT4-Turbo Scores \ DeepSeek-Coder Scores	0	1	2	3	4	5	6	7	8	9	10
0	4234	709	292	733	162	772	615	757	549	208	238
1	2216	561	179	335	46	189	108	161	177	39	72
2	1399	395	236	415	97	172	126	149	125	26	39
3	4281	1041	1048	2203	714	1754	1216	1509	1396	215	403
4	3856	602	939	2315	915	3580	3658	5566	3315	799	719
5	173	55	38	137	49	145	118	170	131	26	48
6	3479	375	581	2010	913	4503	5877	10088	6122	2039	1317
7	3531	395	395	2127	952	4844	7032	13651	9087	4048	2999
8	1618	107	106	755	379	2277	3363	7757	5434	3330	2506
9	550	26	12	171	79	519	956	2577	1937	1637	1032
10	4	0	2	0	1	3	3	17	27	24	23

Distribution of DeepSeek-V2-Chat Scores

GPT4-Turbo Scores \ DeepSeek-V2-Chat Scores	0	1	2	3	4	5	6	7	8	9	10
0	4734	289	616	590	575	76	309	690	969	370	51
1	946	290	641	601	450	41	205	372	375	146	16
2	355	104	436	745	776	41	235	252	165	65	5
3	1067	120	493	1964	5682	491	2368	2019	1231	323	22
4	942	54	115	551	6030	1544	6528	5696	3633	1150	21
5	60	4	21	22	174	25	213	309	197	62	3
6	1059	39	72	160	2132	863	7165	12538	9840	3395	41
7	1615	44	76	96	676	223	3826	14028	19145	9157	175
8	616	21	17	12	70	24	512	3725	10809	11584	242
9	48	0	1	0	3	4	14	273	1598	7385	170
10	3	0	0	0	0	0	0	2	3	74	22

Figure 12. Comparison of quality scores from three oracles. Score distributions from individual oracles are presented along the top and right edges of the heatmap.
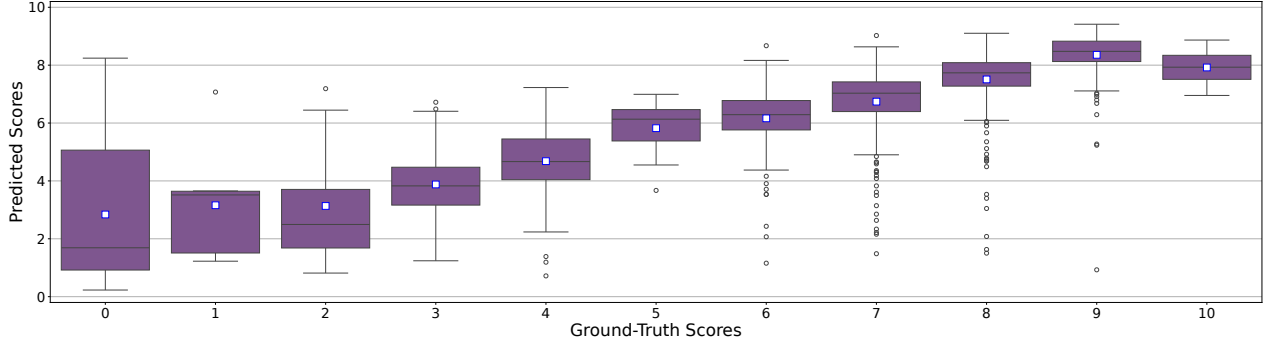
Figure 13. Boxplot of predicted scores from the quality filter corresponding to each individual ground-truth score. Means are indicated with small white squares in the bars.

A.2.3. Evaluation of Quality Scorer

The quality scorer is designed to filter out low-quality code files from the dataset. Once a threshold is set, the scorer effectively functions as a classifier. To ensure flexibility in threshold selection, the quality scorer must provide accurate predictions across arbitrary score intervals. During training, we used mean squared error (MSE) between predicted scores and ground-truth values as the loss function for the regression model. However, as shown in Figure 12, the ground-truth scores are not uniformly distributed, potentially leading to varying accuracy across different intervals under MSE-loss. To further assess the quality filter, we evaluated the mean absolute error (MAE) for samples at each ground-truth score. Figure 13 shows the predictions of the quality scorer over $1,178$ test samples, categorized by the ground-truth scores from DeepSeek-V2-Chat. While identifying zero-condition cases remains challenging, the overall MAE across categories is $\epsilon_{cMAE} = 1.37$. In

contrast, the MAE computed directly across all samples is $\epsilon_{MAE} = 0.91$. The formulations for computing ϵ_{cMAE} and ϵ_{MAE} are

$$\epsilon_{cMAE} = \frac{1}{11} \sum_{i=0}^{10} \left(\frac{1}{|C_i|} \sum_{j \in C_i} |\hat{y}_j - i| \right), \quad \epsilon_{MAE} = \frac{\sum_{i=0}^{10} \sum_{j \in C_i} |\hat{y}_j - i|}{\sum_{i=0}^{10} |C_i|},$$

where C_i is the subset of samples with ground-truth score i and $\hat{y}_j$ is the predicted fractional score of sample j. Both metrics indicate that the scorer performs reliably.