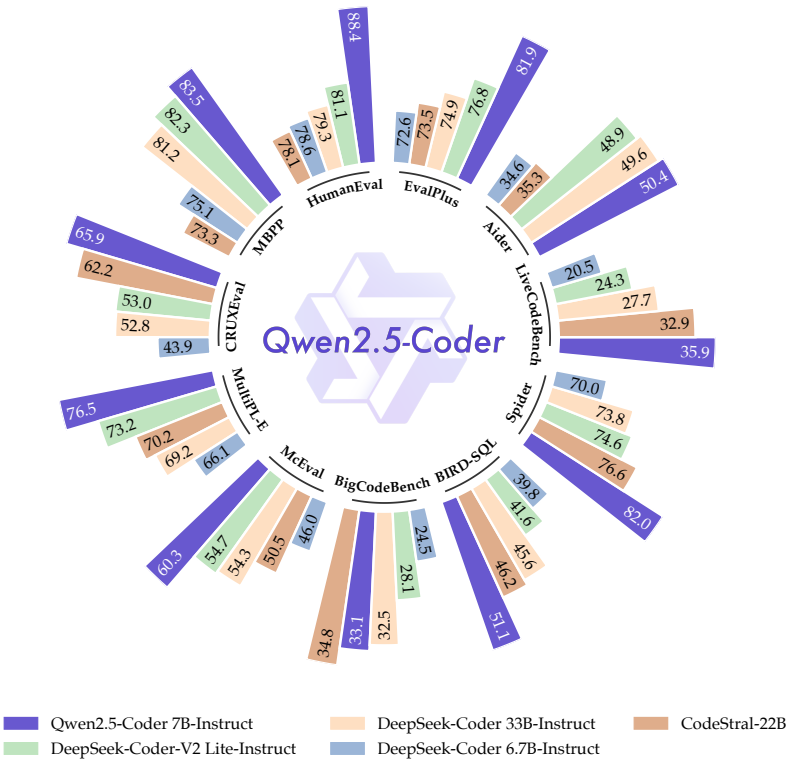
Qwen2.5-Coder Technical Report

Binyuan Hui* **Jian Yang*** **Zeyu Cui*** **Jiaxi Yang***
Dayiheng Liu Lei Zhang Tianyu Liu Jiajun Zhang Bowen Yu Kai Dang An Yang
Rui Men Fei Huang Xingzhang Ren Xuancheng Ren Jingren Zhou Junyang Lin†
Qwen Team **Alibaba Group**

https://hf.co/Qwen/Qwen2.5-Coder-7B-Instruct
https://github.com/QwenLM/Qwen2.5-Coder

Abstract

In this report, we introduce the Qwen2.5-Coder series, a significant upgrade from its predecessor, CodeQwen1.5. This series includes two models: Qwen2.5-Coder-1.5B and Qwen2.5-Coder-7B. As a code-specific model, Qwen2.5-Coder is built upon the Qwen2.5 architecture and continues pretrained on a vast corpus of over 5.5 trillion tokens. Through meticulous data cleaning, scalable synthetic data generation, and balanced data mixing, Qwen2.5-Coder demonstrates impressive code generation capabilities while retaining general versatility. The model has been evaluated on a wide range of code-related tasks, achieving state-of-the-art (SOTA) performance across more than 10 benchmarks, including code generation, completion, reasoning, and repair, consistently outperforming larger models of the same model size. We believe that the release of the Qwen2.5-Coder series will not only push the boundaries of research in code intelligence but also, through its permissive licensing, encourage broader adoption by developers in real-world applications.



*Equal core contribution, †Corresponding author

1 Introduction

With the rapid development of large language models (LLMs) (Brown, 2020; Achiam et al., 2023; Touvron et al., 2023; Dubey et al., 2024; Jiang et al., 2023; Bai et al., 2023b; Yang et al., 2024; Anthropic, 2024; OpenAI, 2024), code-specific language models have garnered significant attention in the community. Built upon pretrained LLMs, code LLMs such as the StarCoder series (Li et al., 2023; Lozhkov et al., 2024), CodeLlama series (Roziere et al., 2023), DeepSeek-Coder series (Guo et al., 2024), CodeQwen1.5 (Bai et al., 2023a), and Code-Stral (team, 2024), have demonstrated superior performance in coding evaluations (Chen et al., 2021; Austin et al., 2021; Cassano et al., 2022; Jain et al., 2024). However, in comparison with the recently state-of-the-art proprietary LLMs, Claude-3.5-Sonnet (Anthropic, 2024) and GPT-4o (OpenAI, 2024), the code LLMs are still falling behind, either open-source or proprietary models.

Building upon our previous work, CodeQwen1.5, we are excited to introduce **Qwen2.5-Coder**, a new series of language models designed to achieve top-tier performance in coding tasks at various model sizes. Qwen2.5-Coder models are derived from the Qwen2.5 LLMs, inheriting their advanced architecture and tokenizer. These models are pretrained on extensive datasets and further fine-tuned on carefully curated instruction datasets specifically designed for coding tasks. The series includes models with 1.5B and 7B parameters, along with their instruction-tuned variants. We are committed to fostering research and innovation in the field of code LLMs, coding agents, and coding assistant applications. Therefore, we are open-sourcing the Qwen2.5-Coder models to the community to support and accelerate advancements in these areas.

Significant efforts have been dedicated to constructing a large-scale, coding-specific pretraining dataset comprising over 5.5 trillion tokens. This dataset is sourced from a broad range of public code repositories, such as those on GitHub, as well as large-scale web-crawled data containing code-related texts. We have implemented sophisticated procedures to recall and clean potential code data and filter out low-quality content using weak model based classifiers and scorers. Our approach encompasses both file-level and repository-level pretraining to ensure comprehensive coverage. To optimize performance and balance coding expertise with general language understanding, we have carefully curated a data mixture that includes code, mathematics, and general texts. To transform models into coding assistants for downstream applications, we have developed a well-designed instruction-tuning dataset. This dataset includes a wide range of coding-related problems and solutions, sourced from real-world applications and synthetic data generated by code-focused LLMs, covering a broad spectrum of coding tasks.

This report introduces the Qwen2.5-Coder series, an upgraded version of CodeQwen1.5, featuring two models: Qwen2.5-Coder-1.5B and Qwen2.5-Coder-7B. Built on the Qwen2.5 architecture and pretrained on over 5.5 trillion tokens, these code-specific models demonstrate exceptional code generation capabilities while maintaining general versatility. Through rigorous data processing and training techniques, Qwen2.5-Coder achieves state-of-the-art performance across more than 10 code-related benchmarks, outperforming larger models in various tasks. The release of these models aims to advance code intelligence research and promote widespread adoption in real-world applications, facilitated by permissive licensing.

2 Model Architecture

Architecture The architecture of Qwen2.5-Coder is the same as Qwen2.5. Table 1 shows the architecture of Qwen2.5-Coder for two different model sizes: 1.5B and 7B parameters. Both sizes share the same architecture in terms of layers, having 28 layers and a head size of 128. However, they differ in several key aspects. The 1.5B model has a hidden size of 1,536, while the 7B model has a much larger hidden size of 3,584. The 1.5B model uses 12 query heads and 2 key-value heads, whereas the 7B model uses 28 query heads and 4 key-value heads, reflecting its larger capacity. The intermediate size also scales with model size, being 8,960 for the 1.5B model and 18,944 for the 7B model. Additionally, the 1.5B model employs

embedding tying, while the 7B model does not. Both models share the same vocabulary size of 151,646 tokens and have been trained on 5.5 trillion tokens.

Tokenization Qwen2.5-Coder inherits the vocabulary from Qwen2.5 but introduces several special tokens to help the model better understand code. Table 2 presents an overview of the special tokens added during training to better capture different forms of code data. These tokens serve specific purposes in the code-processing pipeline. For instance, <|endoftext|> marks the end of a text or sequence, while the <|fim_prefix|>, <|fim_middle|>, and <|fim_suffix|> tokens are used to implement the Fill-in-the-Middle (FIM) (Bavarian et al., 2022) technique, where a model predicts the missing parts of a code block. Additionally, <|fim_pad|> is used for padding during FIM operations. Other tokens include <|repo_name|>, which identifies repository names, and <|file_sep|>, used as a file separator to better manage repository-level information. These tokens are essential in helping the model learn from diverse code structures and enable it to handle longer and more complex contexts during both file-level and repo-level pretraining.

Configuration	Qwen2.5-Coder 1.5B	Qwen2.5-Coder 7B
Hidden Size	1,536	3,584
# Layers	28	28
# Query Heads	12	28
# KV Heads	2	4
Head Size	128	128
Intermediate Size	8,960	18,944
Embedding Tying	True	False
Vocabulary Size	151,646	151,646
# Trained Tokens	5.5T	5.5T

Table 1: Architecture of Qwen2.5-Coder.

Token	Token ID	Description
<|endoftext|>	151643	end of text/sequence
<|fim_prefix|>	151659	FIM prefix
<|fim_middle|>	151660	FIM middle
<|fim_suffix|>	151661	FIM suffix
<|fim_pad|>	151662	FIM pad
<|repo_name|>	151663	repository name
<|file_sep|>	151664	file separator

Table 2: Overview of the special tokens.

3 Pre-training

3.1 Pretraining Data

Large-scale, high-quality, and diverse data forms the foundation of pre-trained models. To this end, we constructed a dataset named Qwen2.5-Coder-Data. This dataset comprises five key data types: Source Code Data, Text-Code Grounding Data, Synthetic Data, Math Data, and Text Data. In this section, we provide a brief overview of the sources and cleaning methods applied to these datasets.

3.1.1 Data Composition

Source Code We collected public repositories from GitHub created before February 2024, spanning 92 programming languages. Similar to StarCoder2 (Lozhkov et al., 2024) and DS-Coder (Guo et al., 2024), we applied a series of rule-based filtering methods. In addition

to raw code, we also collected data from Pull Requests, Commits, Jupyter Notebooks, and Kaggle datasets, all of which were subjected to similar rule-based cleaning techniques.

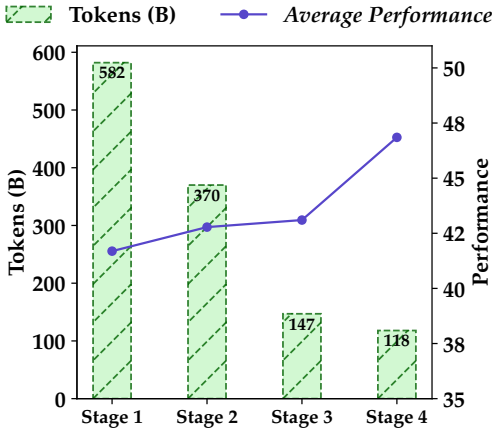


Figure 1: Number of data tokens across different cc-stages, and the validation effectiveness of training Qwen2.5-Coder using corresponding data.

Text-Code Grounding Data We curated a large-scale and high-quality text-code mixed dataset from Common Crawl, which includes code-related documentation, tutorials, blogs, and more. Instead of the conventional URL-based multi-stage recall method, we developed a coarse-to-fine hierarchical filtering approach for raw data. This method offers two key advantages:

1. It enables precise control over each filter's responsibility, ensuring comprehensive handling of each dimension.
2. It naturally assigns quality scores to the dataset, with data retained in the final stage being of higher quality, providing valuable insights for quality-driven data mixing.

We designed a cleaning pipeline for the Text-Code Grounding Data, where each filter level is built using smaller models, such as fastText. Although we experimented with larger models, they did not yield significant benefits. A likely explanation is that smaller models focus more on surface-level features, avoiding unnecessary semantic complexity.

In Qwen2.5-Coder, we applied this process iteratively. As shown in Figure 1, each iteration resulted in improvement. Through 4-stage filtering, the average scores on HumanEval and MBPP increased from 41.6% to 46.8% compared to the baseline, demonstrating the value of high-quality Text-Code Grounding Data for code generation.

Synthetic Data Synthetic data offers a promising way to address the anticipated scarcity of training data. We used CodeQwen1.5, the predecessor of Qwen2.5-Coder, to generate large-scale synthetic datasets. To mitigate the risk of hallucinations during this process, we introduced an executor for validation, ensuring that only executable code was retained.

Math Data To enhance the mathematical capabilities of Qwen2.5-Coder, we integrated the pre-training corpus from Qwen2.5-Math into the Qwen2.5-Coder dataset. Importantly, the inclusion of mathematical data did not negatively impact the model's performance on code tasks. For further details on the collection and cleaning process, please refer to the Qwen2.5-Math technical report.

Text Data Similar to the Math Data, we included high-quality general natural language data from the pre-training corpus of the Qwen2.5 model to preserve Qwen2.5-Coder's general capabilities. This data had already passed stringent quality checks during the

cleaning phase of Qwen2.5's dataset, so no further processing was applied. However, all code segments were removed from the general Text data to avoid overlap with our code data, ensuring the independence of different data sources.

3.1.2 Data Mixture

Balancing Code, Math, and Text data is crucial for building a robust foundational model. Although the research community has explored this balance before, there is limited evidence regarding its scalability to large datasets. To address this, we conducted empirical experiments with different ratios of Code, Math, and Text data, designing multiple experiments to identify an optimal combination rapidly. Specifically, as shown in Table 3, we compared three different Code: Text ratios — 100:0:0, 85:10:5, and 70:20:10.

Interestingly, we found that the 7:2:1 ratio outperformed the others, even surpassing the performance of groups with a higher proportion of code. A possible explanation is that Math and Text data may positively contribute to code performance, but only when their concentration reaches a specific threshold. In future work, we plan to explore more efficient ratio mechanisms and investigate the underlying causes of this phenomenon. Ultimately, we selected a final mixture of 70% Code, 20% Text, and 10% Math. The final training dataset comprises 5.2 trillion tokens.

Token Ratio			Coding		Math		General			*Average*
Code	Text	Math	Common	BCB	MATH	GSM8K	MMLU	CEval	HellaSwag	
100	0	0	**49.8**	**40.3**	10.3	23.8	42.8	35.9	58.3	31.3
85	15	5	43.3	36.2	26.1	52.5	56.8	57.1	70.0	48.9
70	20	10	48.3	38.3	**33.2**	**64.5**	**62.9**	**64.0**	**73.5**	**55.0**

Table 3: The performance of Qwen2.5-Coder training on different data mixture policy.

3.2 Training Policy

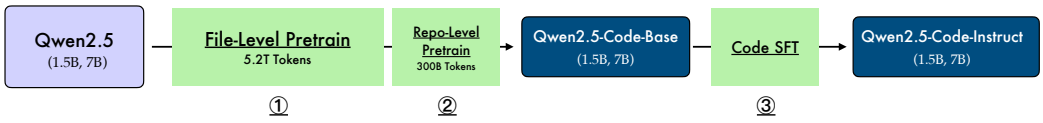


Figure 2: The three-stage training pipeline for Qwen2.5-Coder.

As shown in 2, we employed a three-stage training approach to train Qwen2.5-Coder, including file-level pretraining, repo-level pretraining, and instruction tuning.

3.2.1 File-Level Pretraining

File-level pretraining focuses on learning from individual code files. In this stage, the maximum training sequence length is set to 8,192 tokens, covering 5.2T of high-quality data. The training objectives include next token prediction and fill-in-the-middle (FIM) (Bavarian et al., 2022). The specific FIM format is shown in Figure 3.

File-Level FIM format.

```
<|fim_prefix|>{code_pre}<|fim_suffix|>{code_suf}<|fim_middle|>{code_mid}<|endoftext|>
```

Figure 3: File-Level FIM format.

3.2.2 Repo-Level Pretraining

After file-level pretraining, we turn to repo-level pretraining, aimed at enhancing the model's long-context capabilities. In this stage, the context length is extended from 8,192 tokens to 32,768 tokens, and RoPE's base frequency is adjusted from 10,000 to 1,000,000. To further leverage the model's extrapolation potential, we applied the YARN mechanism (Peng et al., 2023), enabling the model to handle sequences up to 131,072 (132K) tokens.

In this stage, we used a large amount of high-quality, long-code data ($\approx$ 300B) and extended file-level FIM to the repo-level FIM followed by methods described in Lozhkov et al. (2024), with the specific format shown in Figure 4.

Repo-Level FIM format.

```
<|repo_name|>{repo_name}
<|file_sep|>{file_path1}
{file_content1}
<|file_sep|>{file_path2}
{file_content2}
<|file_sep|>{file_path3}
<|fim_prefix|>{code_pre}<|fim_suffix|>{code_suf}<|fim_middle|>{code_fim}<|endoftext|>
```

Figure 4: Repo-Level FIM format.

4 Post-training

4.1 A Recipe for Instruction Data

Multilingual Programming Code Identification We fine-tune a CodeBERT to perform the language identification model to categorize documents into nearly 100 programming languages. We keep the instruction data of the mainstream programming languages and randomly discard a portion of the instruction data of the long-tail languages. If a given sample contains very little code data or even no code snippets, the sample will possibly be classified into "No Programming Language" tag. We remove most of the samples without code snippets to keep the code generation capability of our instruction model.

Instruction Synthesis from GitHub For the unsupervised data (code snippets) massively existing in many websites (e.g. GitHub), we try to construct the supervised instruction dataset. Specifically, we use the LLM to generate the instruction from the code snippets within 1024 tokens and then we use the code LLM to generate the response Sun et al. (2024). Finally, we use the LLM scorer to filter the low-quality ones to obtain the final pair. Given the code snippets of different programming languages, we construct an instruction dataset from the code snippets. To increase the diversity of the instruction dataset. Conversely, we first generate the answers from the code. Then we use the LLM scorer to filter the low-quality to obtain the final triplet. Similarly, given the code snippets of different programming languages, we can construct an instruction dataset with the universal code from the code snippets. To fully unleash the potential of our proposed method, we also include the open-source instruction dataset in the seed instruction dataset. Finally, we combine three parts instruction dataset for supervised fine-tuning.

Multilingual Code Instruction Data To bridge the gap among different programming languages, we propose a multilingual multi-agent collaborative framework to synthesize the multilingual instruction corpora. We introduce language-specific agents, where a set of specialized agents are created and each dedicated to a particular programming language. These agents are initialized with language-specific instruction data derived from the limited

existing multilingual instruction corpora. The multilingual data generation process can be split into: (1) Language-Specific Intelligent Agents: We create a set of specialized agents, each dedicated to a particular programming language. These agents are initialized with language-specific instruction data derived from curated code snippets. (2) Collaborative Discussion Protocol: Multiple language-specific agents engage in a structured dialogue to formulate new instructions and solutions. This process can result in either enhancing existing language capabilities or generating instructions for a novel programming language. (3) Adaptive Memory System: Each agent maintains a dynamic memory bank that stores its generation history to avoid generating the similar samples. (4) Cross-Lingual Discussion: We implement a novel knowledge distillation technique that allows agents to share insights and patterns across language boundaries, fostering a more comprehensive understanding of programming concepts. (5) Synergy Evaluation Metric: We develop a new metric to quantify the degree of knowledge sharing and synergy between different programming languages within the model. (6) Adaptive Instruction Generation: The framework includes a mechanism to dynamically generate new instructions based on identified knowledge gaps across languages.

Checklist-based Scoring for Instruction Data To completely evaluate the quality of the created instruction pair, we introduce several scoring points for each sample: (1) Question&Answer Consistency: Whether Q&A are consistent and correct for fine-tuning. (2) Question&Answer Relevance: Whether Q&A are related to the computer field. (3) Question&Answer Difficulty: Whether Q&A are sufficiently challenging. (4) Code Exist: Whether the code is provied in question or answer. (5) Code Correctness: Evaluate whether the provided code is free from syntax errors and logical flaws. (6) Consider factors like proper variable naming, code indentation, and adherence to best practices. (7) Code Clarity: Assess how clear and understandable the code is. Evaluate if it uses meaningful variable names, proper comments, and follows a consistent coding style. (8) Code Comments: Evaluate the presence of comments and their usefulness in explaining the code's functionality. (9) Easy to Learn: determine its educational value for a student whose goal is to learn basic coding concepts. After gaining all scores $(s_1, \ldots, s_n)$, we can get the final score with $s = w_1 s_1 + \cdots + w_n s_n$, where $(w_1, \ldots, w_n)$ are a series of pre-defined weights.

A multilingual sandbox for code verification To further verify the correctness of the code syntax, we use the code static checking for all extracted code snippets of programming languages (e.g. Python, Java, and C++). We parse the code snippet into the abstract syntax tree and filter out the code snippet, where the parsed nodes in code snippet have parsing errors. We create a multilingual sandbox to support the code static checking for the main programming language. Further, the multilingual sandbox is a comprehensive platform designed to validate code snippets across multiple programming languages. It automates the process of generating relevant unit tests based on language-specific samples and evaluates whether the provided code snippets can successfully pass these tests. Especially, only the self-contained (e.g. algorithm problems) code snippet will be fed into the multilingual sandbox. The multilingual verification sandbox is mainly comprised of five parts:

1. **Language Support Module:**
   - Implements support for multiple languages (e.g., Python, Java, C++, JavaScript)
   - Maintains language-specific parsing and execution environments
   - Handles syntax and semantic analysis for each supported language
2. **Sample Code Repository:**
   - Stores a diverse collection of code samples for each supported language
   - Organizes samples by language, difficulty level, and programming concepts
   - Regularly updated and curated by language experts
3. **Unit Test Generator:**
   - Analyzes sample code to identify key functionalities and edge cases
   - Automatically generates unit tests based on the expected behavior
   - Produces test cases covering various input scenarios and expected outputs

4. **Code Execution Engine:**
   - Provides isolated environments for executing code snippets securely
   - Supports parallel execution of multiple test cases
   - Handles resource allocation and timeout mechanisms
5. **Result Analyzer:**
   - Compares the output of code snippets against expected results from unit tests
   - Generates detailed reports on test case successes and failures
   - Provides suggestions for improvements based on failed test cases

4.2 Training Policy

Coarse-to-fine Fine-tuning We first synthesized tens of millions of low-quality but diverse instruction samples to fine-tune the base model. In the second stage, we adopt millions of high-quality instruction samples to improve the performance of the instruction model with rejection sampling and supervised fine-tuning. For the same query, we use the LLM to generate multiple candidates and then use the LLM to score the best one for supervised fine-tuning.

Mixed Tuning Since most instruction data have a short length, we construct the instruction pair with the FIM format to keep the long context capability of the base model. Inspired by programming language syntax rules and user habits in practical scenarios, we leverage the `tree-sitter-languages`[1] to parse the code snippets and extract the basic logic blocks as the middle code to infill. For example, the abstract syntax tree (AST) represents the structure of Python code in a tree format, where each node in the tree represents a construct occurring in the source code. The tree's hierarchical nature reflects the syntactic nesting of constructs in the code and includes various elements such as expressions, statements, and functions. By traversing and manipulating the AST, we can randomly extract the nodes of multiple levels and use the code context of the same file to uncover the masked node. Finally, we optimize the instruction model with a majority of standard SFT data and a small part of FIM instruction samples.

5 Decontamination

To ensure that Qwen2.5-Coder does not produce inflated results due to test set leakage, we performed decontamination on all data, including both pre-training and post-training datasets. We removed key datasets such as HumanEval, MBPP, GSM8K, and MATH. The filtering was done using a 10-gram overlap method, where any training data with a 10-gram string-level overlap with the test data was removed.

6 Evaluation on Base Models

For the base model, we conducted a comprehensive and fair evaluation in six key aspects, including code generation, code completion, code reasoning, mathematical reasoning, general natural language understanding and long-context modeling. To ensure the reproducibility of all results, we made all evaluation codes publicly available[2]. For comparing models, we chose the most popular and powerful open source language models, including the StarCoder2 and DeepSeek-Coder series. Below is the list of artifacts used in the evaluation for this section.

6.1 Code Generation

HumanEval and MBPP Code generation serves as a fundamental capability for code models to handle more complex tasks. We selected two popular code generation benchmarks

[1] https://pypi.org/project/tree-sitter-languages/
[2] https://github.com/QwenLM/Qwen2.5-Coder

Artifact	Public link
Qwen2.5-Coder-1.5B	https://hf.co/qwen/Qwen/Qwen2.5-Coder-1.5B
Qwen2.5-Coder-7B	https://hf.co/qwen/Qwen/Qwen2.5-Coder-7B
CodeQwen1.5-7B-Base	https://huggingface.co/Qwen/CodeQwen1.5-7B
StarCoder2-3B	https://hf.co/bigcode/starcoder2-3b
StarCoder2-7B	https://hf.co/bigcode/starcoder2-7b
StarCoder2-15B	https://hf.co/bigcode/starcoder2-15b
DS-Coder-1.3B-Base	https://hf.co/deepseek-ai/deepseek-coder-1.3b-base
DS-Coder-6.7B-Base	https://hf.co/deepseek-ai/deepseek-coder-6.7b-base
DS-Coder-33B-Base	https://hf.co/deepseek-ai/deepseek-coder-33b-base
DS-Coder-V2-Lite-Base	https://hf.co/deepseek-ai/DeepSeek-Coder-V2-Lite-Base
DS-Coder-V2-Base	https://hf.co/deepseek-ai/DeepSeek-Coder-V2-Base

Table 4: All artifacts released and used in this section.

Model	Size	HumanEval		MBPP			BigCodeBench	
		HE	*HE+*	*MBPP*	*MBPP+*	*3-shot*	*Full*	*Hard*
1B+ Models								
StarCoder2-3B	3B	31.7	27.4	60.2	49.1	-	21.4	4.7
DS-Coder-1.3B	1.3B	34.8	26.8	55.6	46.9	46.2	26.1	3.4
Qwen2.5-Coder-1.5B	1.5B	**43.9**	**36.6**	**69.2**	**58.6**	**59.2**	**34.6**	**9.5**
6B+ Models								
StarCoder2-7B	7B	35.4	29.9	54.4	45.6	-	27.7	8.8
StarCoder2-15B	15B	46.3	37.8	66.2	53.1	-	38.4	12.2
DS-Coder-6.7B-Base	6.7B	47.6	39.6	70.2	56.6	60.6	41.1	11.5
DS-Coder-V2-Lite-Base	2.4/16B	40.9	34.1	71.9	59.4	-	30.6	8.1
CodeQwen1.5-7B-Base	7B	51.8	45.7	72.2	60.2	61.8	45.6	15.6
Qwen2.5-Coder-7B-Base	7B	**61.6**	**53.0**	**76.9**	**62.9**	**68.8**	**45.8**	**16.2**
20B+ Models								
DS-Coder-33B-Base	33B	54.9	47.6	74.2	60.7	66.0	49.1	20.3

Table 5: Performance of various models on HumanEval, MBPP and the "complete" task of BigCodeBench.

to evaluate Qwen2.5-Coder, namely HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021). HumanEval consists of 164 manually written programming tasks, each providing a Python function signature and a docstring as input to the model. MBPP, on the other hand, comprises 974 programming problems created by crowdsource contributors. Each problem includes a problem statement (i.e., a docstring), a function signature, and three test cases.

To further ensure accurate evaluation, EvalPlus (Liu et al., 2023) extends HumanEval into HumanEval+ by adding 80 times more unique test cases and correcting inaccurate ground-truth solutions in HumanEval. Similarly, MBPP+ offers 35 times more test cases than the original MBPP.

Additionally, we should notice that MBPP 3-shot is particularly suitable for monitoring model convergence during training. Early in the convergence process, the model tends to be unstable, causing significant fluctuation in metrics, and simple 3-shot examples effectively mitigate it. Therefore, we also report the results of MBPP 3-shot performance.

As shown in Table 5, Qwen2.5-Coder have shown impressive performance in basic code generation, achieving state-of-the-art results among open-source models of the same size and surpassing even larger models. In particular, Qwen2.5-Coder-7B-Base outperforms the previous best dense model, DS-Coder-33B-Base, across all five metrics.

BigCodeBench-Complete BigCodeBench (Zhuo et al., 2024) is a recent and more challenging benchmark for code generation, primarily aimed at evaluating the ability of tool-use and complex instruction following. The base model generates the expected code through a

Model	Size	Python	C++	Java	PHP	TS	C#	Bash	JS	Average
1B+ Models										
StarCoder2-3B	3B	31.7	30.4	29.8	32.9	39.6	34.8	13.9	35.4	31.1
DS-Coder-1.3B-Base	1.3B	34.8	31.1	32.3	24.2	28.9	36.7	10.1	28.6	28.3
Qwen2.5-Coder-1.5B	1.5B	**42.1**	**42.9**	**38.6**	**41.0**	**49.1**	**46.2**	**20.3**	**49.1**	**41.1**
6B+ Models										
StarCoder2-7B	7B	35.4	40.4	38.0	30.4	34.0	46.2	13.9	36.0	34.3
StarCoder2-15B	15B	46.3	47.2	46.2	39.1	42.1	53.2	15.8	43.5	41.7
DS-Coder-6.7B-Base	6.7B	49.4	50.3	43.0	38.5	49.7	50.0	28.5	48.4	44.7
DS-Coder-V2-Lite-Base	2.4/16B	40.9	45.9	34.8	47.2	48.4	41.7	19.6	44.7	40.4
CodeQwen1.5-7B-Base	7B	51.8	52.2	42.4	46.6	52.2	55.7	36.7	49.7	48.4
Qwen2.5-Coder-7B-Base	7B	**61.6**	**62.1**	**53.2**	**59.0**	**64.2**	**60.8**	**38.6**	**60.3**	**57.5**
20B+ Models										
DS-Coder-33B-Base	33B	56.1	58.4	51.9	44.1	52.8	51.3	32.3	55.3	50.3

Table 6: Performance of different models on MultiPL-E.

completion mode, given a function signature and documentation, which is referred to as BigCodeBench-Complete. It consists of two subsets: the full set and the hard set. Compared to HumanEval and MBPP, BigCodeBench is suited for out-of-distribution (OOD) evaluation.

Table 5 illustrates that Qwen2.5-Coder continues to show strong performance on BigCodeBench-Complete, underscoring the model's generalization potential.

Multi-Programming Language The evaluations mentioned above focus on the Python language. However, we expect a strong code model to be not only proficient in Python but also versatile across multiple programming languages to meet the complex and evolving demands of software development. To more comprehensively evaluate Qwen2.5-Coder's proficiency in handling multiple programming languages, we selected the MultiPL-E (Cassano et al., 2022) and choose to evaluate eight mainstream languages from these benchmark, including Python, C++, Java, PHP, TypeScript, C#, Bash and JavaScript.

As shown in the table 6, Qwen2.5-Coder also achieved state-of-the-art results in the multi-programming language evaluation, with its capabilities well-balanced across various languages. It scored over 60% in five out of the eight languages.

6.2 Code Completion

HumanEval Infilling Many developer aid tools rely on the capability to autocomplete code based on preceding and succeeding code snippets. Qwen2.5-Coder utilizes the Fill-In-the-Middle (FIM) training strategy, as introduced in Bavarian et al. (2022), enabling the model to generate code that is contextually coherent. To assess its code completion proficiency, we utilize the HumanEval Infilling benchmark Allal et al. (2023). This benchmark challenges the model to accurately predict missing sections of code within tasks derived from HumanEval. We use the single-line infilling settings across Python, Java, and JavaScript, focusing on predicting a single line of code within given contexts. Performance was measured using the Exact Match metric, which determines the proportion of the first generated code line that precisely match the ground truth.

The table 7 illustrates that Qwen2.5-Coder surpasses alternative models concerning model size. Specifically, **Qwen2.5-Coder-1.5B** achieves an average performance improvement of 3.7%, rivaling the majority of models exceeding 6 billion parameters. Moreover, **Qwen2.5-Coder-7B-Base** stands as the leading model among those over 6 billion parameters, matching the performance of the formidable 33 billion parameter model, DS-Coder-33B-Base. Notably, we excluded DS-Coder-v2-236B from comparison due to its design focus not being on code completion tasks.

Model	Size	HumanEval-Infilling			
		Python	*Java*	*JavaScript*	*Average*
1B+ Models					
StarCoder2-3B	3B	70.9	84.4	81.8	79.0
DS-Coder-1.3B-Base	1.3B	72.8	84.3	81.7	79.6
Qwen2.5-Coder-1.5B	1.5B	**77.0**	**85.6**	**85.0**	**82.5**
6B+ Models					
StarCoder2-7B	7B	70.8	86.0	84.4	80.4
StarCoder2-15B	15B	78.1	87.4	84.1	81.3
DS-Coder-6.7B-Base	6.7B	78.1	87.4	84.1	83.2
DS-Coder-V2-Lite-Base	2.4/16B	78.7	87.8	85.9	84.1
CodeQwen1.5-7B-Base	7B	75.8	85.7	85.0	82.2
Qwen2.5-Coder-7B-Base	7B	**79.7**	**88.5**	**87.6**	**85.3**
20B+ Models					
DS-Coder-33B-Base	33B	80.1	89.0	86.8	85.3

Table 7: Performance of different approaches on the HumanEval Infilling Tasks

6.3 Code Reasoning

Code is a highly abstract form of logical language, and reasoning based on code helps us determine whether a model truly understands the reasoning flow behind the code. We selected CRUXEval (Gu et al., 2024) as the benchmark, which includes 800 Python functions along with corresponding input-output examples. It consists of two distinct tasks: CRUXEval-I, where the large language model (LLM) must predict the output based on a given input; and CRUXEval-O, where the model must predict the input based on a known output. For both CRUXEval-I and CRUXEval-O, we used a chain-of-thought (CoT) approach, requiring the LLM to output steps sequentially during simulated execution.

As shown in Table 8, Qwen2.5-Coder delivered highly promising results, achieving a score of 56.5 on CRUXEval-I and 56.0 on CRUXEval-O, thanks to our focus on executable quality during the code cleaning process.

Model	Size	CRUXEval	
		Input-CoT	*Output-CoT*
1B+ Models			
StarCoder2-3B	3B	42.1	29.2
DS-Coder-1.3B-Base	1.3B	32.1	28.2
Qwen2.5-Coder-1.5B	1.5B	**43.8**	**34.6**
6B+ Models			
StarCoder2-7B	7B	39.5	35.1
StarCoder2-15B	15B	46.1	47.6
DS-Coder-6.7B-Base	6.7B	39.0	41.0
DS-Coder-V2-Lite-Base	2.4/16B	53.4	46.1
CodeQwen1.5-7B-Base	7B	44.8	40.1
Qwen2.5-Coder-7B-Base	7B	**56.5**	**56.0**
20B+ Models			
DS-Coder-33B-Base	33B	50.6	48.8
DS-Coder-V2-Base	21/236B	62.7	67.4

Table 8: Performance of different models on CRUXEval with *Input-CoT* and *Output-CoT* settings.

6.4 Math Reasoning

Mathematics and coding have always been closely intertwined. Mathematics forms the foundational discipline for coding, while coding serves as a vital tool in mathematical fields. As such, we expect an open and powerful code model to exhibit strong mathematical capabilities as well. To assess Qwen2.5-Coder's mathematical performance, we selected five popular benchmarks, including MATH (Hendrycks et al., 2021), GSM8K (Cobbe et al., 2021), MMLU-STEM (Hendrycks et al., 2020) and TheoremQA (Chen et al., 2023). As shown in Table 9, Table 3 highlights Qwen2.5-Coder's strengths in mathematics, which likely stem from two key factors: first, the model's strong foundation built on Qwen2.5, and second, the careful mixing of code and mathematical data during training, which has ensured a well-balanced performance across these domains.

Model	Size	MATH *4-shot*	GSM8K *4-shot*	MMLU STEM *5-shot*	TheoremQA *5-shot*
1B+ Models					
StarCoder2-3B	3B	10.8	21.6	34.9	12.1
DS-Coder-1.3B-Base	1.3B	4.6	4.4	24.5	8.9
Qwen2.5-Coder-1.5B	1.5B	**30.9**	**65.8**	**49.0**	**21.4**
6B+ Models					
StarCoder2-7B	7B	14.6	32.7	39.8	16.0
StarCoder2-15B	15B	23.7	57.7	49.2	20.5
DS-Coder-6.7B-Base	6.7B	10.3	21.3	34.2	13.6
DS-Coder-V2-Lite-Base	2.4/16B	39.0	67.1	58.5	29.3
CodeQwen1.5-7B-Base	7B	10.6	37.7	39.6	15.8
Qwen2.5-Coder-7B-Base	7B	**46.6**	**83.9**	**67.6**	**34.0**
20B+ Models					
DS-Coder-33B-Base	33B	14.4	35.4	39.5	17.5

Table 9: Performance of various models on four math benchmark, named MATH, GSM8K, MMLU STEM and TheoremQA respectively.

Model	Size	MMLU Base	MMLU Redux
1B+ Models			
StarCoder2-3B	3B	36.6	37.0
DS-Coder-1.3B-Base	1.3B	25.8	24.5
Qwen2.5-Coder-1.5B	1.5B	**53.6**	**50.9**
6B+ Models			
StarCoder2-7B	7B	38.8	38.6
StarCoder2-15B	15B	64.1	48.8
DS-Coder-6.7B-Base	6.7B	36.4	36.5
DS-Coder-V2-Lite-Base	2.4/16B	60.5	58.3
CodeQwen1.5-7B-Base	7B	40.5	41.2
Qwen2.5-Coder-7B-Base	7B	**68.0**	**66.6**
20B+ Models			
DS-Coder-33B-Base	33B	39.4	38.7

Table 10: MMLU results of different models, a general benchmark for common knowledge.

Model	Size	ARC-Challenge	TruthfulQA	WinoGrande	HellaSwag
1B+ Models					
StarCoder2-3B	3B	34.2	40.5	57.1	48.1
DS-Coder-1.3B-Base	1.3B	25.4	42.7	53.3	39.5
Qwen2.5-Coder-1.5B	1.5B	**45.2**	**44.0**	**60.7**	**61.8**
6B+ Models					
StarCoder2-7B	7B	38.7	42.0	57.1	52.4
StarCoder2-15B	15B	47.2	37.9	64.3	64.1
DS-Coder-6.7B-Base	6.7B	36.4	40.2	57.6	53.8
DS-Coder-V2-Lite-Base	2.4/16B	57.3	38.8	72.9	-
CodeQwen1.5-7B-Base	7B	35.7	42.2	59.8	56.0
Qwen2.5-Coder-7B-Base	7B	**60.9**	**50.6**	**72.9**	**76.8**
20B+ Models					
DS-Coder-33B-Base	33B	42.2	40.0	62.0	60.2

Table 11: General performance of different models on four popular general benchmark, ARC-Challenge, TruthfulQA, WinoGrande and HellaSwag.

6.5 General Natural Language

In addition to mathematical ability, we aim to retain as much of the base model's general-purpose capabilities as possible, such as general knowledge. To evaluate general natural language understanding, we selected MMLU (Hendrycks et al., 2021) and its variant MMLU-Redux (Gema et al., 2024), along with four other benchmarks: ARC-Challenge (Clark et al., 2018), TruthfulQA (Lin et al., 2021), WinoGrande (Sakaguchi et al., 2019), and HellaSwag (Zellers et al., 2019). Similar to the results in mathematics, Table 11 highlights Qwen2.5-Coder's advantage in general natural language capabilities compared to other coders, further validating the effectiveness of Qwen2.5-Coder data mixing strategy.

6.6 Long-Context Evaluation

Long context capability is crucial for code LLMs, serving as the core skill for understanding repository-level code and becoming a code agent. However, most of current code models still have very limited support for length, which hinders their potential for practical application. Qwen2.5-Coder aims to further advance the progress of open-source code models in long context modeling. To achieve this, we have collected and constructed long sequence code data at the repository level for pre-training. Through careful data proportioning and organization, we have enabled it to support input lengths of up to 128K tokens.

Needle in the Code We created a simple but basic synthetic task called *Needle in the Code*, inspired by popular long-context evaluations in the text domain. In this task, we inserted a very simple custom function at various positions within a code repo (we chose Megatron [3] to honor its contributions to open-source LLMs!) and tested whether the model could replicate this function at the end of the codebase. The figure below shows that Qwen2.5-Coder is capable of successfully completing this task within a 128k length range.

7 Evaluation on Instruct Models

For the evaluation of the instruct models, we rigorously assessed six core areas: *code generation*, *code reasoning*, *code editing*, *text-to-sql*, *mathematical reasoning* and *general natural language understanding*. The evaluation was structured to ensure a fair and thorough comparison across models. All evaluation code is publicly accessible for reproducibility[4]. To ensure a broad comparison, we included some of the most popular and widely-used open-source

[3] https://github.com/NVIDIA/Megatron-LM

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

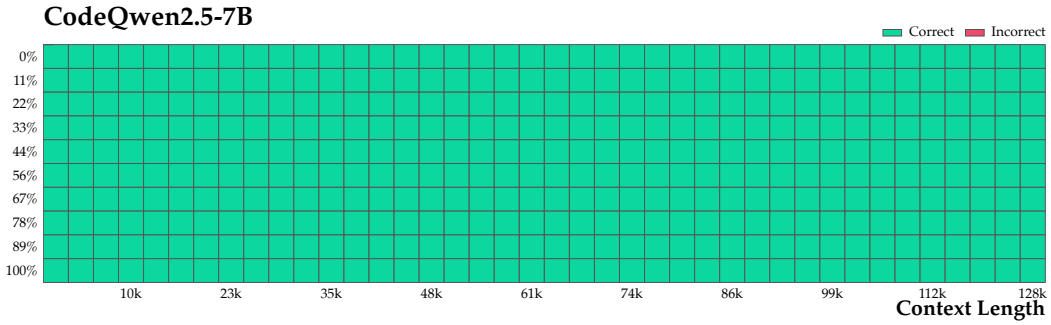


Figure 5: The long context ability of Qwen2.5-Coder, evaluated by Needle in Code.

instruction-tuned models, notably versions from the DeepSeek-Coder series and Codestral models. Below is a list of all artifacts referenced in this section.

Artifact	Public link
Qwen2.5-Coder-1.5B-Instruct	https://hf.co/qwen/Qwen/Qwen2.5-Coder-1.5B-Instruct
Qwen2.5-Coder-7B-Instruct	https://hf.co/qwen/Qwen/Qwen2.5-Coder-7B-Instruct
CodeQwen1.5-7B-Chat	https://huggingface.co/Qwen/CodeQwen1.5-7B-Chat
CodeStral-22B	https://hf.co/mistralai/Codestral-22B-v0.1
DS-Coder-1.3B-instruct	https://hf.co/deepseek-ai/deepseek-coder-1.3b-instruct
DS-Coder-6.7B-instruct	https://hf.co/deepseek-ai/deepseek-coder-6.7b-instruct
DS-Coder-33B-instruct	https://hf.co/deepseek-ai/deepseek-coder-33b-instruct
DS-Coder-V2-Lite-Instruct	https://hf.co/deepseek-ai/DeepSeek-Coder-V2-Lite-Instruct
DS-Coder-V2-Instruct	https://hf.co/deepseek-ai/DeepSeek-Coder-V2-Instruct

Table 12: All artifacts released and used in this section.

7.1 Code Generation

Building on the performance improvements of the Qwen2.5-Coder series base models, our Qwen2.5-Coder series instruct models similarly demonstrated outstanding performance in code generation tasks.

HumanEval and MBPP We also evaluated the code generation capabilities of the Qwen2.5-Coder series instruct models using the EvalPlus (Liu et al., 2023) dataset. As illustrated by the experimental results in Table 13, our Qwen2.5-Coder-7B-Instruct model demonstrated superior accuracy, significantly outperforming other models with a comparable number of parameters. Remarkably, it even exceeded the performance of larger models with over 20 billion parameters, such as CodeStral-22B and DS-Coder-33B-Instruct. Notably, Qwen2.5-Coder-7B-Instruct was the only model in our evaluation to surpass an 80% accuracy rate on HumanEval+, achieving an impressive 84.1%.

BigCodeBench-Instruct The *instruct* split provided by BigCodeBench (Zhuo et al., 2024) is intended for assessing the code generation abilities of instruct models. We assessed the Qwen2.5-Coder series instruct models on the BigCodeBench-Instruct. As shown in Table 13, the Qwen2.5-Coder-7B-Instruct outperformed other instruct models with similar parameter sizes, achieving higher accuracy scores on both the full and hard subsets, reaching 41.0% on the full subset and 18.2% on the hard subset, demonstrating the Qwen2.5-Coder series instruct models' powerful code generation capabilities.

LiveCodeBench LiveCodeBench (Jain et al., 2024) is a comprehensive and contamination-free benchmark designed to evaluate the coding capabilities of LLMs. It continuously

Model	Size	HumanEval		MBPP		BigCodeBench		LiveCodeBench
		HE	*HE+*	*MBPP*	*MBPP+*	*Full*	*Hard*	*Pass@1*
1B+ Models								
DS-Coder-1.3B-Instruct	1.3B	65.2	61.6	61.6	52.6	22.8	3.4	9.3
Qwen2.5-Coder-1.5B-Instruct	1.5B	**70.7**	**66.5**	**69.2**	**59.4**	**32.5**	**6.8**	**15.7**
6B+ Models								
DS-Coder-6.7B-Instruct	6.7B	78.6	70.7	75.1	66.1	35.5	10.1	20.5
DS-Coder-V2-Lite-Instruct	2.4/16B	81.1	75.0	82.3	68.8	36.8	16.2	24.3
CodeQwen1.5-7B-Chat	7B	86.0	79.3	83.3	71.4	39.6	17.2	20.1
Qwen2.5-Coder-7B-Instruct	7B	**88.4**	**84.1**	**83.5**	**71.7**	**41.0**	**18.2**	**37.6**
20B+ Models								
CodeStral-22B	22B	78.1	74.4	73.3	68.2	47.1	20.6	32.9
DS-Coder-33B-Instruct	33B	79.3	68.9	81.2	70.1	46.5	17.6	27.7

Table 13: The performance of different instruct models on code generation by HumanEval, MBPP, bigcodebench and livecodebench. For bigcodebench here, we report "instruct" tasks score.

Model	Size	Python	Java	C++	C#	TS	JS	PHP	Bash	Average
1B+ Models										
DS-Coder-1.3B-Instruct	1.3B	65.2	51.9	45.3	55.1	59.7	52.2	45.3	12.7	48.4
Qwen2.5-Coder-1.5B-Instruct	1.5B	**71.2**	**55.7**	**50.9**	**64.6**	**61.0**	**62.1**	**59.0**	**29.1**	**56.7**
6B+ Models										
DS-Coder-6.7B-Instruct	6.7B	78.6	68.4	63.4	72.8	67.2	72.7	68.9	36.7	66.1
DS-Coder-V2-Lite-Instruct	2.4/16B	81.1	76.6	75.8	76.6	80.5	77.6	74.5	43.0	73.2
CodeQwen1.5-7B-Chat	7B	83.5	70.9	72	75.9	76.7	77.6	73.9	41.8	71.6
Qwen2.5-Coder-7B-Instruct	7B	**87.8**	**76.5**	**75.6**	**80.3**	**81.8**	**83.2**	**78.3**	**48.7**	**76.5**
20B+ Models										
CodeStral-22B	22B	78.1	71.5	71.4	77.2	72.3	73.9	69.6	47.5	70.2
DS-Coder-33B-Instruct	33B	79.3	73.4	68.9	74.1	67.9	73.9	72.7	43.0	69.2
DS-Coder-V2-Instruct	21/236B	90.2	82.3	84.8	82.3	83	84.5	79.5	52.5	79.9

Table 14: The performance of different approaches on instruct format MultiPL-E.

gathers new problems from leading competitive programming platforms like LeetCode[5], AtCoder[6], and CodeForces[7], ensuring an up-to-date and diverse set of challenges. Currently, it hosts over 600 high-quality coding problems published between May 2023 and September 2024.

To better demonstrate our model's effectiveness on real-world competitive programming tasks, we conduct evaluation of the Qwen-2.5-Coder series instruct models on the LiveCodeBench (2305-2409) dataset. As illustrated in Table 13, the Qwen-2.5-Coder-7B-Instruct model achieved an impressive Pass@1 accuracy of 37.6%, significantly outperforming other models of comparable parameter scales. Notably, it also surpassed larger models such as CodeStral-22B and DS-Coder-33B-Instruct, underscoring the Qwen-2.5-Coder series' exceptional capabilities in handling complex code generation challenges.

Multi-Programming Language The Qwen2.5-Coder series instruct models have inherited the high performance of the base model on the Multi-Programming Language. To further evaluate their capabilities, we tested the instruct models on two specific benchmarks: MultiPL-E (Cassano et al., 2022) and McEval (Chai et al., 2024).

MultiPL-E As demonstrated by the evaluation results in Table 14, Qwen2.5-Coder-7B-Instruct consistently outperforms other models with the same number of parameters, including DS-Coder-V2-Lite-Instruct, in code generation tasks across eight programming

[5] https://leetcode.com
[6] https://atcoder.jp
[7] https://codeforces.com

languages. With an average accuracy of 76.5%, Qwen2.5-Coder-7B-Instruct surpasses even larger models such as CodeStral-22B and DS-Coder-33B-Instruct (despite having over 20 billion parameters), highlighting its powerful code generation capabilities in multiple programming languages.

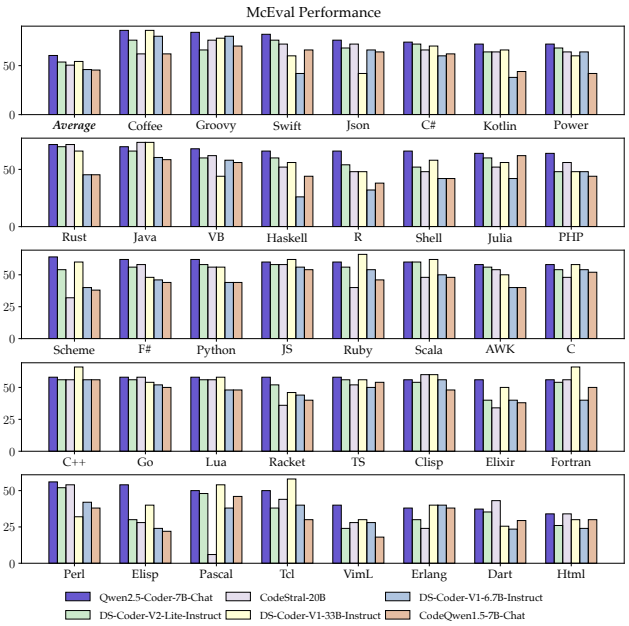


Figure 6: The McEval Performance of Qwen2.5-Coder-7B-Instruct compared with popular open-source large code models with similar size.

McEval To comprehensively assess the code generation capabilities of the Qwen2.5-Coder series models across a broader range of programming languages, we evaluated them on the McEval benchmark (Chai et al., 2024), which spans 40 programming languages and includes 16,000 test cases. As shown in Figure 6, the Qwen2.5-Coder-7B-Instruct model excels when compared to other open-source models on the McEval benchmark, particularly across a wide range of programming languages. Its average accuracy not only exceeds that of much larger models such as DS-Coder-33B-Instruct and CodeStral-22B but also demonstrates a notable advantage over models of comparable parameter size.

7.2 Code Reasoning

To assess the code reasoning capabilities of the Qwen2.5-Coder series instruct models, we performed an evaluation on CRUXEval (Gu et al., 2024). As illustrated by the experimental results in Table 15, the Qwen2.5-Coder-7B-Instruct model achieved Input-CoT and Output-CoT accuracies of 65.8% and 65.9%, respectively. This represents a notable improvement over the DS-Coder-V2-Lite-Instruct model, with gains of 12.8% in Input-CoT accuracy and 13.0% in Output-CoT accuracy. Furthermore, the Qwen2.5-Coder-7B-Instruct model outperformed larger models, such as the CodeStral-22B and DS-Coder-33B-Instruct, underscoring its superior code reasoning capabilities despite its smaller size.

Figure 7 illustrates the relationship between model sizes and code reasoning capabilities. The Qwen2.5-Coder instruct models stand out for delivering superior code reasoning performance with the fewest parameters, surpassing the results of other open-source large language models by a significant margin. According to this trend, we expect that code reasoning performance comparable to GPT-4o could be achieved with a model around the 30 billion parameters scale.

Model	Size	CRUXEval *Input-CoT*	CRUXEval *Output-CoT*
1B+ Models			
DS-Coder-1.3B-Instruct	1.3B	14.8	28.1
Qwen2.5-Coder-1.5B-Instruct	1.5B	**45.4**	**37.5**
6B+ Models			
DS-Coder-6.7B-Instruct	6.7B	42.6	45.1
DS-Coder-V2-Lite-Instruct	2.4/16B	53.0	52.9
CodeQwen1.5-7B-Chat	7B	42.1	38.5
Qwen2.5-Coder-7B-Instruct	7B	**65.8**	**65.9**
20B+ Models			
CodeStral-22B	22B	48.0	60.6
DS-Coder-33B-Instruct	33B	47.3	50.6
DS-Coder-V2-Instruct	21/236B	70.0	75.1

Table 15: The CRUXEval performance of different instruct models, with *Input-CoT* and *Output-CoT* settings.

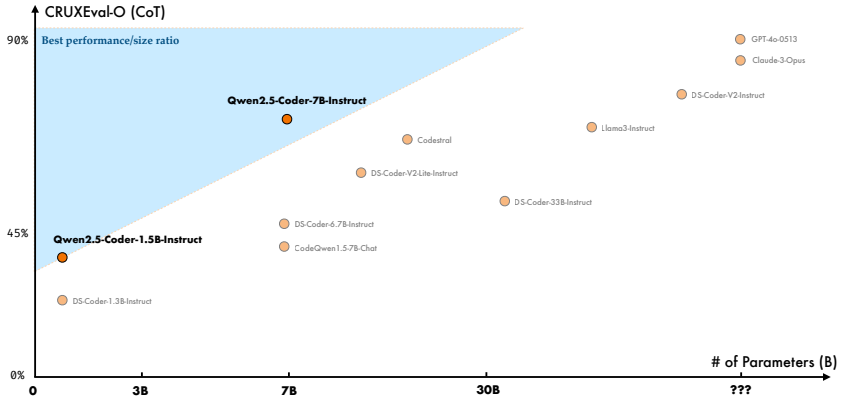


Figure 7: The relationship between model sizes and code reasoning capabilities. The x-axis represents the parameter sizes of different models, and the y-axis indicates the CRUXEval-O (CoT) scores respectively.

7.3 Code Editing

Aider[8] has created a code editing benchmark designed to quantitatively measure its collaboration with large language models (LLMs). Drawing from a set of 133 Python exercises sourced from Exercism[9], the benchmark tests the ability of Aider and LLMs to interpret natural language programming requests and translate them into executable code that successfully passes unit tests. This assessment goes beyond evaluating raw coding proficiency; it also examines how effectively LLMs can edit existing code and format those modifications for seamless integration with Aider's system, ensuring that local source files can be updated without issues. The comprehensive nature of this benchmark reflects both the technical aptitude of the LLMs and their consistency in task completion.

Table 16 highlights the performance of several language models in the Code Editing task. Among them, Qwen2.5-Coder-7B-Instruct demonstrates outstanding code repair capabilities. Despite its relatively modest size of 7 billion parameters, it achieves an impressive

[8] https://github.com/paul-gauthier/aider
[9] https://github.com/exercism/python

Model	Size	Aider	
		Pass@1	*Pass@2*
1B+ Models			
DS-Coder-1.3B-Instruct	1.3B	17.3*	21.1*
Qwen2.5-Coder-1.5B-Instruct	1.5B	**30.1**	**33.1**
6B+ Models			
DS-Coder-6.7B-Instruct	6.7B	34.6*	42.1*
DS-Coder-V2-Lite-Instruct	2.4/16B	48.9*	55.6*
CodeQwen1.5-7B-Chat	7B	23.3	33.1
Qwen2.5-Coder-7B-Instruct	7B	**50.4**	**57.1**
20B+ Models			
CodeStral-22B	22B	35.3*	45.1*
DS-Coder-33B-Instruct	33B	49.6*	59.4*
DS-Coder-V2-Instruct	21/236B	73.7	-

Table 16: The code editing ability of different instruct models evaluated by Aider benchmark. * indicates that the experimental results have been reproduced in our experiments, and the *whole* edit-format was consistently applied across all experiments.

PASS@1 accuracy of 50.4%, significantly outperforming comparable models. Notably, it also surpasses larger models such as CodeStral-22B (22 billion parameters) and DS-Coder-33B-Instruct (33 billion parameters), showcasing its remarkable efficiency and effectiveness in code editing tasks.

7.4 Text-to-SQL

SQL is one of the essential tools in daily software development and production, but its steep learning curve often hinders free interaction between non-programming experts and databases. To address this issue, the Text-to-SQL task was introduced, aiming for models to automatically map natural language questions to structured SQL queries. Previous improvements in Text-to-SQL focused primarily on structure-aware learning, domain-specific pre-training, and sophisticated prompt designs.

Thanks to the use of finely crafted synthetic data during both pre-training and fine-tuning, we significantly enhanced Qwen2.5-Coder's capability in Text-to-SQL tasks. We selected two well-known benchmarks, Spider (Yu et al., 2018) and BIRD (Li et al., 2024), for comprehensive evaluation. To ensure a fair comparison between Qwen2.5-Coder and other open-source language models on this task, we used a unified prompt template as input, following the work of Chang & Fosler-Lussier (2023). As shown in Figure 8, the prompt consists of table representations aligned with database instructions, examples of table content, optional additional knowledge, and natural language questions. This standardized prompt template minimizes biases that may arise from prompt variations. As shown in Figure 9, Qwen2.5-Coder outperforms other code models of the same size on the Text-to-SQL task.

Model	Size	MATH	GSM8K	GaoKao2023en	OlympiadBench	CollegeMath	AIME24
DS-Coder-V2-Lite-Instruct	2.4/16B	61.0	**87.6**	56.1	26.4	39.8	6.7
Qwen2.5-Coder-7B-Instruct	7B	**66.8**	86.7	**60.5**	**29.8**	**43.5**	**10.0**
Model	**Size**	**AMC23**	**MMLU**	**MMLU-Pro**	**IFEval**	**CEval**	**GPQA**
DS-Coder-V2-Lite-Instruct	2.4/16B	40.4	42.5	60.6	38.6	60.1	27.6
Qwen2.5-Coder-7B-Instruct	7B	**42.5**	**45.6**	**68.7**	**58.6**	**61.4**	**35.6**

Table 17: The performance of math and General.

Prompt template for text-to-SQL tasks.

```
CREATE TABLE "list" (
  "LastName" TEXT,
  "FirstName" TEXT,
  "Grade" INTEGER,
  "Classroom" INTEGER,
  PRIMARY KEY(LastName, FirstName)
);
/* 3 example rows:
SELECT * FROM list LIMIT 3;
LastName     FirstName  Grade  Classroom
CAR          MAUDE      2      101
KRISTENSEN   STORMY     6      112
VANDERWOUDE  SHERWOOD   3      107
*/
(...other tables omitted...)
-- External Knowledge: ...
-- Using valid SQLite and understanding External Knowledge, answer the following
     questions for the tables provided above.
Question: How many students are there?

SELECT count(*) FROM list;
```

Figure 8: Prompt template of Qwen2.5-Coder for text-to-SQL tasks.

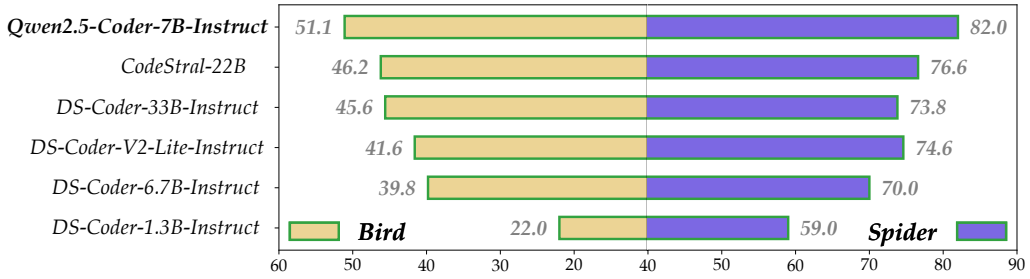


Figure 9: The text-to-SQL evaluation on various instruct code models.

7.5 Math Reasoning and General Natural Language

In this section, we present a comparative analysis of the performance between our Qwen2.5-Coder-7B-Instruct model and the DS-Coder-V2-Lite-Instruct model, focusing on both mathematical computation and general natural language processing tasks. As indicated in Table 17, the Qwen2.5-Coder-7B-Instruct model outperforms the DS-Coder-V2-Lite-Instruct in 11 out of 12 tasks. This result underscores the model's versatility, excelling not only in complex coding tasks but also in sophisticated general tasks, thus distinguishing it from its competitors.

8 Conclusion

This work introduces Qwen2.5-Coder, the latest addition to the Qwen series. Built upon Qwen2.5, a top-tier open-source LLM, Qwen2.5-Coder has been developed through extensive pre-training and post-training of Qwen2.5-1.5B and Qwen2.5-7B on large-scale datasets. To ensure the quality of the pre-training data, we have curated a dataset by collecting public code data and extracting high-quality code-related content from web texts, while

filtering out low-quality data using advanced classifiers. Additionally, we have constructed a meticulously designed instruction-tuning dataset to transform the base code LLM into a strong coding assistant.

Looking ahead, our research will focus on exploring the impact of scaling up code LLMs in terms of both data size and model size. We will also continue to enhance the reasoning capabilities of these models, aiming to push the boundaries of what code LLMs can achieve.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

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

Anthropic. Claude 3.5 sonnet. https://www.anthropic.com/news/claude-3-5-sonnet, 2024. 2024.06.21.

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

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023a.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023b.

Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. Efficient training of language models to fill in the middle. *arXiv preprint arXiv:2207.14255*, 2022.

Tom B Brown. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

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

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

Shuaichen Chang and Eric Fosler-Lussier. How to prompt llms for text-to-sql: A study in zero-shot, single-domain, and cross-domain settings. *arXiv preprint arXiv:2305.11853*, 2023.

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

Wenhu Chen, Ming Yin, Max Ku, Pan Lu, Yixin Wan, Xueguang Ma, Jianyu Xu, Xinyi Wang, and Tony Xia. Theoremqa: A theorem-driven question answering dataset. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 7889–7901, 2023.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.