ExecRepoBench: Multi-level Executable Code Completion Evaluation

Anonymous ACL submission

Abstract

Code completion has become an essential tool for daily software development. Existing evaluation benchmarks often employ static methods that do not fully capture the dynamic nature of real-world coding environments and face significant challenges, including limited context length, reliance on superficial evaluation metrics, and potential overfitting to training datasets. In this work, we introduce a novel framework for enhancing code completion in software development through the creation of a repository-level benchmark ExecRepoBench and the instruction corpora Repo-Instruct, aim at improving the functionality of open-source large language models (LLMs) in real-world coding scenarios that involve complex interdependencies across multiple files. ExecRepoBench include 1.2K samples from active Python repositories. Plus, we present a multi-level grammar-based completion methodology conditioned on the abstract syntax tree to mask code fragments at various logical units (e.g. statements, expressions, and functions). Then, we fine-tune the open-source LLM with 7B parameters on Repo-Instruct to produce a strong code completion baseline model Qwen2.5-Coder-Instruct-C based on the open-source model. Qwen2.5-Coder-Instruct-C is evaluated on ExecRepoBench, which gets both competitive results on code generation and code completion. The deployment of Qwen2.5-Coder-Instruct-C can be used as a high-performance, local service for programming development[1].

1 Introduction

In the field of software engineering, the emergence of large language models (LLMs) designed specifically for code-related tasks has represented a significant advancement. These code LLMs (Li et al., 2022; Allal et al., 2023), such as DeepSeek-Coder (Guo et al., 2024a) and Qwen-Coder (Hui et al., 2024), have been pre-trained on extensive datasets comprising billions of code-related data. The advent of code LLMs has revolutionized the automation of software development tasks, providing contextually relevant code suggestions and facilitating code generation.

[1]The evaluation code and dataset will be released

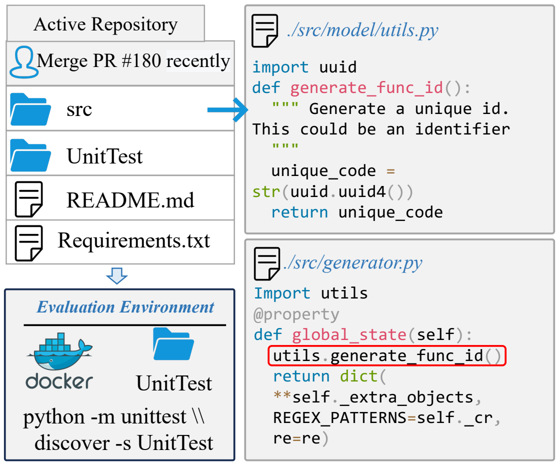


Figure 1: Executable Repository-level code evaluation with the given test cases.

The code completion task holds paramount importance in modern software development, acting as a cornerstone for enhancing coding efficiency and accuracy. By analyzing the context of the ongoing work and using sophisticated algorithms to predict and suggest the next segments of code, code completion tools drastically reduce the time and effort programmers spend on writing boilerplate code, navigating large codebases, or recalling complex APIs and frameworks, which both accelerates the software development cycle and significantly diminishes the likelihood of syntax errors and bugs, leading to cleaner, more maintainable code. The recent code LLMs (Bavarian et al., 2022; Zheng et al., 2023) complete the middle code based on the prefix and suffix code through prefix-suffix-middle (PSM) and suffix-prefix-middle (SPM) pre-training paradigm. To correctly evaluate the code completion capability, the HumanEval benchmark (Allal et al., 2023; Zheng et al., 2023) is extended to the infilling task by randomly masking some code spans and lines and prompting LLms to predict the middle code. The recent works (Ding et al., 2023b, 2022, 2023a) propose to use the cross-file context to complete the current file and then score the results with n-gram string match. *However, the community still lacks an executable evaluation repository-level benchmark from live repositories and the corresponding instruction corpora.*

In this work, we benchmark, elicit, and enhance code repository-level completion tasks of open-source large language models (LLMs) by creating the repository-level instruction corpora **Repo-Instruct** and the corresponding benchmark **ExecRepoBench** for utilization and evaluation for code completion in real-world software development scenarios, where projects frequently involve complex dependencies across multiple files. Unlike previous benchmarks with text-matching metrics (e.g. exact match (EM) and edit similarity (ES)), in Figure 1, ExecRepoBench is constructed with repository-level unit tests to verify the correctness of the completion code, which contains 1.2K samples from 50 active Python repositories. To facilitate the attention of the community for the code completion task, we propose the multi-level grammar-based completion to create Repo-Instruct, where the code fragments under the different levels of logical units are masked for completion using the parsed abstract syntax tree (AST). During supervised finetuning (SFT), the code snippet of the repository is packed into the instruction data for the code completion LLMs Qwen2.5-Coder-Instruct-C, where the query gives the prefix code of the current file, suffix code of the current file, and code snippets of other files.

Qwen2.5-Coder-Instruct-C is evaluated on the code generation benchmark (Cassano et al., 2023) and our created code completion benchmark ExecRepoBench. The results demonstrate that Qwen2.5-Coder-Instruct-C consistently achieves state-of-the-art performance across all languages, notably surpassing the previous baselines. The contributions are summarized as follows:

- We introduce executable repository-level benchmark ExecRepoBench for code completion evaluation, which collects the active repositories from GitHub and modify them into executable formats with test cases.
- We propose the multi-level grammar-based completion conditioned on the abstract syntax tree, where the statement-level, expression-level, function-level, and class-level code snippets are extracted for multi-level completion instruction corpora Repo-Instruct
- Based on the open-source LLMs and the instruction corpora Repo-Instruct, we fine-tune base LLMs with 7B parameters Qwen2.5-Coder-Instruct-C with a mixture of code completion data and standard instruction corpora, which can be used as a local service for programming developer.

Figure 2: Classification of collected repositories. ExecRepoBench contains 14 main domains of our collected repositories, such as 'Date and Time', 'Data Processing', 'Media&Image', 'Design Patterns'.

2 ExecRepoBench Construction

Data Collection and Annotation The collected and refined repositories should follow the following guidelines: (1) Search Github code repositories of the Python language that have been continuously updated. (2) Given the collected repositories, the annotator should collect or create the test cases for evaluation. (3) All collected repositories should pass the test cases in a limited time for fast evaluation (< 2 minutes). In Figure 2, we collect diverse repositories for comprehensive code completion evaluation. Figure 2 lists 14 main domains of our collected repositories, such as 'Date and Time', 'Data Processing', 'Media&Image', 'Design Patterns', and 'Data Validation'. We feed the prefix tokens and suffix tokens of the current file with the

	Random Completion			Grammar-based Completion		
	Span	Single Line	Multiple Line	Expression	Statement	Function
|Samples|	42	34	38	407	266	377
Context Tokens	0/277.7K/27.7K	333/276.7K/22.6K	0/1484.1K/55.5K	0/1484.4K/36.2K	0/1484.6K/93.1K	0/1484.5K/65.1K
Prefix Tokens	0/32.2K/1.4K	0/38.3K/1.9K	0/7.2K/978.0	0/35.8K/1.5K	0/12.8K/786.0	0/38.3K/2.4K
Middle Tokens	3/22/7.0	4/48/15.0	4/156/40.0	2/150/13.0	2/74/9.0	7/123/33.0
Suffix Tokens	0/7.9K/924.0	0/2.0K/562.0	0/5.8K/935.0	1/39.0K/1.3K	1/40.1K/2.6K	1/37.8K/2.0K
	|Repositories|	|Directories|	|Stars|	|Files|	|Python Files|	|Other Files|
Repository Overview	50	2/115/15	1/39K/2.6K	15/790/113	4/411/38	10/379/75

Table 1: Data statistics of ExecRepoBench.

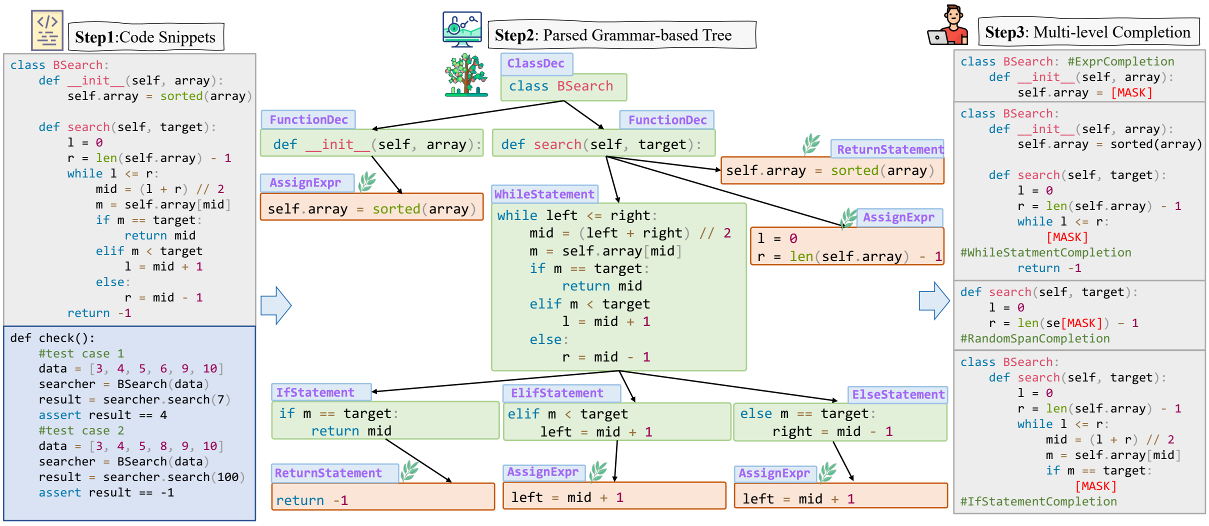


Figure 3: Multi-level Completion based on the parsed abstract syntax tree from the code snippet. We use `tree-sitter` to parse code into abstract syntax trees (ASTs), which allows for the extraction of basic logic blocks at different granularities—expression-level, statement-level, and function-level—to be infilled using the code context from the same repository. The approach also incorporates heuristic completion techniques, specifically random Line completion and random span completion.

context tokens into the LLM to predict the middle code tokens. To avoid data leakage, we remove exact matches (20-gram word overlap) from CrossCodeEval (Ding et al., 2023b) and the pre-training corpus stack V2 (Lozhkov et al., 2024).

Data Statistics To create the benchmark ExecRepoBench, we first construct the random span completion, random single-line completion, and random multi-line completion task by masking contiguous spans and lines of the chosen file of the whole repository. For the grammar-based completion, we first parse the code into an abstract syntax tree (AST) tree and randomly mask the node to match the input habits of programming developers habits. Besides, we sort the context files using the relevance between the current masked file and truncate the tokens exceeding the maximum supported length of the code LLM. Table 1 lists 6 completion types in ExecRepoBench, including random completion (span completion, single-line completion, and multple-line completion) and grammar-based completion (expression completion, statement completion, and function completion).

3 Multi-level Code Completion

3.1 Problem Defintion

In-file Completion Given the code x^{L_k} of the current file of programming language L_k ($L_k \in L_{all} = \{L_{i=1}\}_{k=1}^{K}$), the LLM infills the middle code x_m conditioned on the prefix code x_p and the suffix code x_s as follow:

$$P(x_m^{L_k}) = P(x_m^{L_k}|x_p^{L_k}, x_s^{L_k}; \mathcal{M}) \tag{1}$$

where $x_p^{L_k}$, $x_s^{L_k}$, and $x_m^{L_k}$ are concatenated as the complete code to be executed with the given test cases to verify the correctness of the response.

Repository-level Completion Another more important completion scenario is the repository-level completion. Given the code snippets $z = \{z_{i=1}^{L_k}\}_{i=1}^{N}$ of N other files, the LLMs try to fill the part code of the current file. Based on the prefix code of the current file $x_p^{L_k}$, suffix code of the current file $x_s^{L_k}$, and the code snippets c in other files in the same repository, the LLMs aim at producing the middle code x_m as:

$$P(x_m^{L_k}) = P(x_m^{L_k}|x_p^{L_k}, x_s^{L_k}, c; \mathcal{M}) \quad (2)$$

where the concatenation of $x_p^{L_k}$, $x_s^{L_k}$, and $x_m^{L_k}$ are used for repository-level execution for evaluation. c denotes the concatenation of all other files.

3.2 Multi-level Grammar-based Completion

Inspired by programming language syntax rules and user habits in practical scenarios, we leverage the `tree-sitter-languages`[2] to parse the code snippets and extract the basic logic blocks as the middle code to infill, as shown in Figure 3. 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.

Expression-level Completion In Figure 3, at the expression level, we focus on completing sub-expressions within larger expressions or simple standalone expressions. This might involve filling in operand or operator gaps in binary operations or providing appropriate function arguments.

Statement-level Completion This level targets the completion of individual statements, such as variable assignments, control flow structures (if statements, for loops), and others. The goal is to maintain the logical flow and ensure syntactic correctness.

Function-level Completion At the function level, our approach involves completing entire function bodies or signature infillings. This includes parameter lists, return types, and the internal logic of the functions.

3.3 Heuristic Completion Techniques

To enhance the performance of our AST-based code infilling, we implement heuristic completion techniques to mimic the complementary habits of human users.

[2] https://pypi.org/project/tree-sitter-languages/

Random Line Completion We randomly select lines from the same file or similar files in the dataset to serve as candidates for completion. This process requires additional context-aware filtering to maintain relevance and accuracy.

Random Span Completion Instead of single lines, we randomly select code spans - sequences of lines that represent cohesive logical units. This approach suits larger blocks of code, needing a finer grasp of context and structure for effective completion.

3.4 Hybrid Instruction Tuning

Different from the base model trained with the FIM objective, we fine-tune the LLM with a mixture of the code completion data $(x_p^{L_k}, x_m^{L_k}, x_s^{L_k}) \in D_c = \{D_c^{L_k}\}_{k=1}^{K}$. The code completion training objective is described as:

$$\mathcal{L}_c = -\frac{1}{K}\sum_{k=1}^{K}\mathbb{E}_{D_c^{L_k}}[P(x_m^k|x_p, x_s, c; \mathcal{M}) \quad (3)$$

where the concatenation of $x_p^{L_k}$, $x_s^{L_k}$, and $x_m^{L_k}$ are used for repository-level execution for evaluation. c is the concatenation of all context code snippets in the same repository.

We also adopt the standard instruction data $(q^{L_k}, a^{L_k}) \in D_{q,a} = \{D_{q,a}^{L_k}\}_{k=1}^{K}$. The question-answer instruction tuning on $D_{q,a}$ is calculated by:

$$\mathcal{L}_{qa} = -\frac{1}{K}\sum_{k=1}^{K}\mathbb{E}_{a^{L_k}, q^{L_k} \in D_{q,a}^{L_k}} \log P(q^{L_k}|a^{L_k}; \mathcal{M}) \quad (4)$$

where (q^{L_k}, a^{L_k}) are query and the corresponding response from the dataset $D_{x,y}$, including code generation, code summarization other code-related tasks. We unify the capability of the code completion and question-answer in a single instruction model. The training objective of the hybrid instruction tuning is described as:

$$L_{all} = \mathcal{L}_c + \mathcal{L}_{qa} \quad (5)$$

where $\mathcal{L}_c$ is the code completion objective and $\mathcal{L}_{qa}$ is the question-answering objective.

4 Experiments

4.1 Code LLMs

We evaluate 30+ LLMs with sizes ranging from 0.5B to 30B+ parameters for open-source code large language models and closed-source general LLMs. For general models, we evaluate

GPTs (Brown et al., 2020; OpenAI, 2023) (GPT-3.5-Turbo, GPT4-o) and Claude series (Anthropic, 2023). For code models, we test CodeLlama (Rozière et al., 2023), StarCoder/StarCoder2 (Li et al., 2023; Lozhkov et al., 2024), CodeGeeX (Zheng et al., 2023), OpenCoder (Huang et al., 2024), Qwen-Coder (Hui et al., 2024), DeepSeek-Coder (Guo et al., 2024a), CodeStral (MistralAI, 2024), Yi-Coder[3], CodeGemma (Zhao et al., 2024), and Granite-Coder (Mishra et al., 2024).

4.2 Implementation Details

We extract the repository-level code snippets from `the-stack-V2`[4] and filter the data with heuristic rules (e.g. GitHub stars and file length). We keep the mainstream programming language (Python, C-sharp, Cpp, Java, Javascript, Typescript, Php) and drop other long-tailed languages to obtain nearly 1.5M repositories. Finally, we obtain the instruction dataset Repo-Instruct contains nearly 3M completion samples. We fine-tune the open-source base foundation LLM Qwen2.5-Coder on nearly 3M instruction samples used in Qwen2.5-Coder (Hui et al., 2024) and code completion data (in-file and cross-file completion data). Qwen2.5-Coder-Instruct-C is fine-tuned on `Megatron-LM`[5] with 64 NVIDIA H100 GPUs. The learning rate first increases into 3×10^{-4} with 100 warmup steps and then adopts a cosine decay scheduler. We adopt the Adam optimizer (Kingma and Ba, 2015) with a global batch size of 2048 samples, truncating sentences to 32K tokens.

4.3 Evaluation Metrics

Edit Similarity We compare the generated code and the ground-truth code using edit similarity (ES) to report string-based scores.

Pass@k Similar to the in-file benchmark HumanEval/MBPP, we employ the Pass@k metric (Chen et al., 2021) based on the executable results to get the reliability evaluation results. In this work, we report the greedy Pass@1 score of all LLMs with greedy inference for a fair comparison.

4.4 Evaluation Benchmarks

Code Completion ExecRepoBench is created with the repository-level unit tests to verify the correctness of the completion code, comprised of 1.2K samples from 50 active Python repositories. We separately report the ES score and Pass@1 in the table.

Code Generation Since the mixture training of instruction samples and code completion samples, Qwen2.5-Coder-Instruct-C also supports answering the code-related queries. We adopt MultiPL-E (Cassano et al., 2023) for general question answering.

4.5 Main Results

ExecRepoBench Table 2 presents a comparative analysis of various code completion models, highlighting their performance across different metrics and parameter sizes. Code LLMs (e.g. CodeLlama and StarCoder) are evaluated across several completion tasks: random completion (span, single-line, multi-line), and grammar-based completion (expression, statement, function). Our proposed model Qwen2.5-Coder-Instruct-C, significantly outperforms competing models in all categories despite having only 7B parameters. Compared to the base foundation model Qwen2.5-Coder and DS-Coder, Qwen2.5-Coder-Instruct-C enhanced by the multi-level grammar-based fine-tuning achieves an impressive average score of 44.2, marking a substantial advancement in the field of code completion technologies. From the table, we can see that there exists a mismatch between the n-gram-based metric ES and execution-based metric pass@1. Granite-Coder-8B gets a good pass@1 score but a bad ES score, which emphasizes the importance of execution-based metric pass@k for correctly evaluating the code completion capability of code LLMs. ES metric has its own inherent flaws, where the score is calculated by the comparison between the generated code and ground-truth code.

Code Generation Table 3 showcases the evaluation results in terms of Pass@1 performance (%) across various models on the MultiPL-E benchmark, focusing on different programming languages. The comparison is categorically divided between proprietary models, like GPT-3.5 and GPT-4, and open-source models, which include DS-Coder, Yi-Coder, and Qwen2.5-Coder variants, among others. o1-preview, a proprietary model, leads with an average of 85.3%, showcasing the difference in performance capability between proprietary and open-source models. The results highlight the effectiveness of our method, particularly in optimizing performance within the constraints of param-

[3]https://huggingface.co/01-ai/Yi-Coder-9B

[4]https://huggingface.co/datasets/bigcode/the-stack-v2

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

Models	Params	Random Completion						Grammar-based Completion						Avg.	
		Span		Single-line		Multi-line		Expression		Statement		Function			
		ES	Pass@1	ES	Pass@1	ES	Pass@1	ES	Pass@1	ES	Pass@1	ES	Pass@1	ES	Pass@1
Code-Llama	7B	3.7	11.9	6.8	35.3	17.2	26.3	5.8	28.5	5.9	23.3	17.3	15.6	9.9	22.7
Code-Llama	13B	3.4	19.0	6.5	35.3	16.7	26.3	5.7	29.5	6.3	25.6	17.4	17.5	9.9	24.4
Code-Llama	34B	4.5	9.5	6.5	32.4	16.9	18.4	6.7	28.7	6.5	22.9	17.8	16.7	10.5	22.6
Code-Llama	70B	3.9	16.7	6.8	38.2	17.7	26.3	5.8	28.7	6.1	25.6	17.6	19.9	10.0	24.9
Codestral	22B	3.5	16.7	9.0	41.2	18.1	28.9	5.7	27.0	6.1	24.8	17.4	16.7	10.0	23.3
StarCoder	1B	31.9	23.8	23.5	41.2	34.8	21.1	19.8	36.6	27.5	25.6	22.8	19.6	23.6	27.7
StarCoder	3B	16.6	11.9	11.7	41.2	24.6	26.3	12.2	28.3	18.0	26.7	19.6	15.1	16.5	23.4
StarCoder	7B	41.6	31.0	30.5	47.1	40.2	28.9	27.6	43.5	31.9	35.3	29.8	22.0	30.3	33.8
StarCoder2	3B	2.9	14.3	5.8	38.2	15.1	21.1	4.8	26.3	5.3	21.4	15.0	15.1	8.5	21.3
StarCoder2	7B	3.1	21.4	5.6	35.3	15.1	23.7	4.9	25.8	5.2	22.9	14.9	15.1	8.5	21.7
StarCoder2	15B	2.7	19.0	5.6	41.2	14.8	23.7	4.9	27.5	5.2	23.7	15.1	15.6	8.5	22.8
DS-Coder	1.3B	35.0	21.4	25.7	47.1	32.1	28.9	27.2	32.9	14.9	27.1	27.6	14.1	24.9	25.3
DS-Coder	6.7B	40.2	28.6	41.0	50.0	47.5	39.5	45.7	37.3	36.1	33.8	45.9	15.1	43.3	29.5
DS-Coder	33B	47.1	33.3	52.0	64.7	50.1	44.7	46.3	40.8	37.8	37.2	49.5	17.0	45.7	32.8
DS-Coder-V2-Lite	2.4/16B	33.0	31.0	44.1	52.9	42.5	39.5	42.4	37.6	30.0	32.3	42.7	16.2	39.4	29.7
Granite-Coder	3B	35.0	21.4	25.7	47.1	32.1	28.9	27.2	32.9	14.9	27.1	27.6	14.1	24.9	25.3
Granite-Coder	8B	0.0	19.0	0.0	58.8	2.6	28.9	5.2	36.6	0.0	26.3	0.0	21.5	1.9	29.1
Granite-Coder	20B	3.2	16.7	7.8	35.3	14.9	23.7	5.2	26.3	5.7	21.4	15.8	15.1	9.1	21.4
Granite-Coder	34B	3.1	16.7	8.0	35.3	15.2	26.3	5.3	26.3	6.2	24.1	15.4	15.6	9.1	22.3
CodeQwen1.5	7B	13.5	16.7	13.8	41.2	17.6	26.3	9.8	26.0	11.3	24.4	14.9	13.0	12.3	21.6
Qwen2.5-Coder	0.5B	11.3	16.7	10.4	47.1	13.0	26.3	10.7	26.0	14.6	24.1	16.2	14.1	13.5	22.0
Qwen2.5-Coder	1.5B	3.5	14.3	3.2	29.4	7.9	15.8	4.0	21.9	3.5	16.9	9.1	11.7	5.6	17.2
Qwen2.5-Coder	3B	13.8	19.0	14.9	44.1	12.5	21.1	13.9	28.0	11.2	23.7	18.5	13.5	14.8	22.3
Qwen2.5-Coder	7B	7.8	16.7	10.2	35.3	12.4	23.7	5.3	24.3	8.1	21.1	11.0	12.5	8.3	19.8
Qwen2.5-Coder	14B	7.0	16.7	12.7	35.3	12.1	18.4	11.4	27.5	15.1	27.8	18.5	13.5	14.4	22.6
Qwen2.5-Coder	32B	3.9	16.7	32.5	47.1	20.3	23.7	21.2	29.5	26.1	33.5	33.0	15.4	25.8	25.7
OpenCoder	1.5B	1.7	11.9	3.4	38.2	5.4	23.7	3.2	26.3	3.2	27.4	6.6	15.4	4.3	22.8
OpenCoder	8B	2.7	14.3	4.4	32.4	10.8	21.1	4.0	29.5	3.7	24.1	7.8	16.7	5.3	23.4
Yi-Coder	1.5B	3.9	16.7	6.6	32.4	16.4	28.9	6.1	25.3	6.4	26.3	17.2	14.1	10.0	21.9
Yi-Coder	9B	3.4	16.7	6.8	29.4	17.7	26.3	5.8	28.0	6.3	26.3	17.6	17.8	10.1	23.9
CodeGemma	2B	21.3	21.4	23.5	38.2	19.8	18.4	24.1	23.6	28.3	21.1	23.4	12.5	24.6	19.6
CodeGemma	7B	12.4	19.0	14.3	35.3	28.2	26.3	15.4	29.2	18.7	36.5	25.8	18.3	19.8	27.1
Qwen2.5-Coder-Instruct-C	7B	75.8	38.1	68.0	41.2	60.2	28.9	76.4	58.7	78.7	45.5	63.9	30.2	72.1	44.2

Table 2: Completion evaluation results of base foundation model and Qwen2.5-Coder-Instruct-C on ExecRepoBench. We report multi-level score for code completion, focusing on expression-level (sub-expressions or standalone expressions), statement-level (individual statements like assignments or control flow structures), and function-level (entire function bodies or signatures) completion. It also introduces heuristic techniques, including random span and line completion, both requiring context-aware filtering to ensure relevance and accuracy.

eter size. Notably, our method, Qwen2.5-Coder-Instruct-C, with 7 billion parameters, outperforms other models in this parameter range across all listed programming languages, achieving an average Pass@1 performance of 76.4%.

5 Analysis

Ablation Study Figure 4 emphasizes the essence of each component in our method by conducting the ablation study. CrossCodeEval (Ding et al., 2023b) is developed using a variety of real-world, openly available repositories with permissive licenses, covering four widely used programming languages: Python, Java, TypeScript, and C-sharp. Figure 4(a) shows the model results of the code completion task CrossCodeEval and Figure 4(b) plots the results on the instruction-following code benchmark MultiPL-E. By unifying the code generation and completion in the same model, Qwen2.5-Coder-Instruct-C can support multiple scenarios.

Case Study Figure 5 showcases a part of a Python module named `BankOperation` which focuses on simulating basic bank account operations. The module, assumed to be spread across files, includes the BankAccount class definition housed within the given code snippet. Within this class, methods are defined for initializing an account (`__init__`), depositing money (deposit), and displaying the account balance (display_balance). The core segment provided adds a withdraw method to this class, which allows for deducting a specified amount from the account's balance if the amount is positive and does not exceed the available balance. Each transaction (deposit and withdrawal) is followed up with a call to `A.sync()`, hinting at an operation to synchronize the current state of the account with a database, potentially managed by

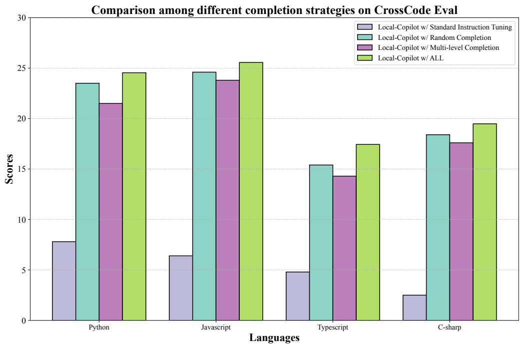
Model	Size	HE	HE+	MBPP	MBPP+	Python	Java	C++	C#	TS	JS	PHP	Bash	Avg.
Closed-APIs														
Claude-3.5-Sonnet-20240620	🔒	89.0	81.1	87.6	72.0	89.6	86.1	82.6	85.4	84.3	84.5	80.7	48.1	80.2
Claude-3.5-Sonnet-20241022	🔒	92.1	86.0	91.0	74.6	93.9	86.7	88.2	87.3	88.1	91.3	82.6	52.5	83.8
GPT-4o-mini-2024-07-18	🔒	87.8	84.8	86.0	72.2	87.2	75.9	77.6	79.7	79.2	81.4	75.2	43.7	75.0
GPT-4o-2024-08-06	🔒	92.1	86.0	86.8	72.5	90.9	83.5	76.4	81.0	83.6	90.1	78.9	48.1	79.1
o1-mini	🔒	97.6	90.2	93.9	78.3	95.7	90.5	93.8	77.2	91.2	92.5	84.5	55.1	85.1
o1-preview	🔒	95.1	88.4	93.4	77.8	96.3	88.0	91.9	84.2	90.6	93.8	90.1	47.5	85.3
0.5B+ Models														
Qwen2.5-Coder-0.5B-Instruct	0.5B	61.6	57.3	52.4	43.7	61.6	57.3	52.4	43.7	50.3	50.3	52.8	27.8	49.6
1B+ Models														
DS-Coder-1.3B-Instruct	1.3B	65.9	60.4	65.3	54.8	65.2	51.9	45.3	55.1	59.7	52.2	45.3	12.7	48.4
Yi-Coder-1.5B-Chat	1.5B	69.5	64.0	65.9	57.7	67.7	51.9	49.1	57.6	57.9	59.6	52.2	19.0	51.9
Qwen2.5-Coder-1.5B-Instruct	1.5B	70.7	66.5	69.2	59.4	71.2	55.7	50.9	64.6	61.0	62.1	59.0	29.1	56.7
3B+ Models														
Qwen2.5-Coder-3B-Instruct	3B	84.1	80.5	73.6	62.4	83.5	74.7	68.3	78.5	79.9	75.2	73.3	43.0	72.1
6B+ Models														
CodeLlama-7B-Instruct	7B	40.9	33.5	54.0	44.4	34.8	30.4	31.1	21.6	32.7	-	28.6	10.1	-
DS-Coder-6.7B-Instruct	6.7B	74.4	71.3	74.9	65.6	78.6	68.4	63.4	72.8	67.2	72.7	68.9	36.7	66.1
CodeQwen1.5-7B-Chat	7B	83.5	78.7	77.7	67.2	84.1	73.4	74.5	77.8	71.7	75.2	70.8	39.2	70.8
Yi-Coder-9B-Chat	9B	82.3	74.4	82.0	69.0	85.4	76.0	67.7	76.6	72.3	78.9	72.1	45.6	71.8
DS-Coder-V2-Lite-Instruct	2.4/16B	81.1	75.6	82.8	70.4	81.1	76.6	75.8	76.6	80.5	77.6	74.5	43.0	73.2
Qwen2.5-Coder-7B-Instruct	7B	88.4	84.1	83.5	71.7	87.8	76.5	75.6	80.3	81.8	83.2	78.3	48.7	76.5
OpenCoder-8B-Instruct	8B	83.5	78.7	79.1	69.0	83.5	72.2	61.5	75.9	78.0	79.5	73.3	44.3	71.0
13B+ Models														
CodeLlama-13B-Instruct	13B	40.2	32.3	60.3	51.1	42.7	40.5	42.2	24.0	39.0	-	32.3	13.9	-
Starcoder2-15B-Instruct-v0.1	15B	67.7	60.4	78.0	65.1	68.9	53.8	50.9	62.7	57.9	59.6	53.4	24.7	54.0
Qwen2.5-Coder-14B-Instruct	14B	89.6	87.2	86.2	72.8	89.0	79.7	85.1	84.2	86.8	84.5	80.1	47.5	79.6
20B+ Models														
CodeLlama-34B-Instruct	34B	48.2	40.2	61.1	50.5	41.5	43.7	45.3	31.0	40.3	-	36.6	19.6	-
CodeStral-22B-v0.1	22B	81.1	73.2	78.2	62.2	81.1	63.3	65.2	43.7	68.6	-	68.9	42.4	-
DS-Coder-33B-Instruct	33B	81.1	75.0	80.4	70.1	79.3	73.4	68.9	74.1	67.9	73.9	72.7	43.0	69.2
CodeLlama-70B-Instruct	70B	72.0	65.9	77.8	64.6	67.8	58.2	53.4	36.7	39.0	-	58.4	29.7	-
DS-Coder-V2-Instruct	21/236B	85.4	82.3	89.4	75.1	90.2	82.3	84.8	82.3	83.0	84.5	79.5	52.5	79.9
Qwen2.5-Coder-32B-Instruct	32B	92.7	87.2	90.2	75.1	92.7	80.4	79.5	82.9	86.8	85.7	78.9	48.1	79.4
Qwen2.5-32B-Instruct	32B	87.8	82.9	86.8	70.9	88.4	80.4	81.0	74.5	83.5	82.4	78.3	46.8	76.9
Qwen2.5-72B-Instruct	32B	85.4	79.3	90.5	77.0	82.9	81.0	80.7	81.6	81.1	82.0	77.0	48.7	75.1
Qwen2.5-SynCoder	32B	92.7	87.8	86.2	74.7	92.1	80.4	80.7	81.6	83.0	85.7	77.6	49.4	78.8
Qwen2.5-Coder-Instruct-C	7B	87.2	81.1	81.7	68.5	89.6	77.2	74.5	81.0	83.6	81.4	77.0	46.8	76.4

Table 3: The performance of different instruction LLMs on EvalPlus and MultiPL-E. “HE” denotes the HumanEval, “HE+” denotes the plus version with more test cases, and “MBPP+” denotes the plus version with more test cases.

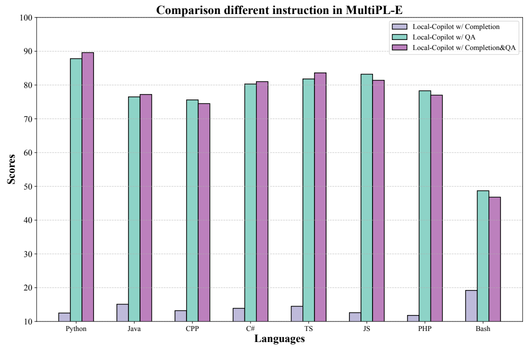
code within `A.py`. Error handling is incorporated within the deposit and withdrawal operations to ensure amounts are valid. The description wraps up this modular approach to implementing a banking system in Python, emphasizing object-oriented programming principles, error management, and database integration. This example shows that Qwen2.5-Coder-Instruct-C can successfully find the dependency from the context file.

6 Related Work

Code Large Language Models In software engineering, the advent of large language models (LLMs) tailored for code-centric tasks has proven to be transformative. Models (Feng et al., 2020; Chen et al., 2021; Scao et al., 2022; Li et al., 2022; Allal et al., 2023; Fried et al., 2022; Wang et al., 2021; Zheng et al., 2024; Jiang et al., 2024; Nijkamp et al., 2023; Wei et al., 2023; Zhao et al., 2024) like CodeLlama (Rozière et al., 2023), DeepSeek-Coder (Guo et al., 2024a), OpenCoder (Huang et al., 2024) and Qwen2.5-Coder (Hui et al., 2024) have fundamentally augmented the development process. These Code LLMs are instrumental in automating repetitive software tasks, proposing code improvements, and facilitating the conversion of natural language into executable code, bringing unique contributions to the enhancement of coding assistance tools.



(a) CrossCodeEval



(b) MultiPl-E

Figure 4: Evaluation results based on standard QA pairs and code completion. By unifying the code generation and completion in the same model, Qwen2.5-Coder-Instruct-C can support multiple scenarios.

[System Prompt]
You are a helpful code completion assistant.
[Question]
##Repo Name##:
BankOperation
##Repo-level Context Code##:
{A.py}
##Prefix Code##:
```
Import A
class BankAccount:
# Initialization method
    def __init__(self, owner, balance=0): 4
        """Initializes the account owner."""
        self.owner = owner
        self.balance = balance
    def deposit(self, amount):
        """increases the account balance."""
        if amount > 0:
            self.balance += amount
            A.sync() #sync database
            return f"current balance is: {self.balance}"
        else:
            return "must be greater than zero."
```
##Suffix Code##:
```
def display_balance(self):
    """Displays the current account balance."""
    return f"current account balance is: {self.balance}"
```
[Answer]
##Middle Code##:
```
def withdraw(self, amount):
    """decreases the account balance."""
    if 0 < amount <= self.balance:
        self.balance -= amount
        A.sync() #sync database
        return f"current balance is: {self.balance}"
    else:
        return "The withdrawal amount is invalid."
```

Figure 5: Example of Qwen2.5-Coder-Instruct-C.

Figure 6: A completion example in instruction format of Qwen2.5-Coder-Instruct-C. This example shows that Qwen2.5-Coder-Instruct-C can find the dependency from the context file in the prompt.

Repo-level Code Evaluation In the domain of code evaluation, a rich tapestry of benchmarks (Zheng et al., 2023; Yu et al., 2024; Yin et al., 2023; Lai et al., 2023) has been woven to address the challenges of accurately assessing code quality and functionality, such as HumanEval/MBPP (Chen et al., 2021; Austin et al., 2021), their upgraded version EvalPlus (Liu et al., 2023a). Realistic scenarios (Liu et al., 2024c,a), such as BigCodeBench (Zhuo et al., 2024), CodeArena (Yang et al., 2024) and SAFIM (Gong et al., 2024), separately evaluate code LLMs for more diverse scenarios and code preferences. An important task FIM (Fried et al., 2022; Bavarian et al., 2022; Ding et al., 2024) is to fill the middle code, given the prefix and suffix code, which provides substantial assistance for software development. Repo-level completion, such as RepoEval (Zhang et al., 2023), CrossCodeEval (Ding et al., 2023b; Wu et al., 2024) and RepoBench (Liu et al., 2023b) only using EM and ES without code execution can not accurately reflect the model performance.

7 Conclusion

In this work, we represent a significant leap forward in the realm of code completion, driven by the advancements in large language models (LLMs) tailored for coding tasks. By introducing an executable repository-level benchmark ExecRepoBench and a multi-level grammar-based instruction corpora Repo-Instruct, we both tackles the limitations of existing benchmarks and set a new standard for evaluating code completion tools in real-world software development scenarios, where the ExecRepoBench is collected from real-world repositories. We adopt expression-level, statement-level, and function-level code completion, along with heuristic methods like random line and span completion to enhance AST-based infilling. The fine-tuned LLM Qwen2.5-Coder-Instruct-C with 7B parameters demonstrates a competitive performance both on code generation and completion.

8 Limitations

We acknowledge the following limitations of this study: (1) The evaluation in repository-level multilingual scenarios are not fully explored. (2) Because the code completion model Qwen2.5-Coder-Instruct-C is only supervised fine-tuned on the 7B open-source base LLMs, we will try different LLM sizes for instruction fine-tuning in the future. (3) The fine-tuned model can be further improved using RLHF for better user experience, such as DPO.

Ethics Statement

This research adheres to ethical guidelines for AI development. We aim to enhance the capabilities of large language models (LLMs) while acknowledging potential risks such as bias, misuse, and privacy concerns. To mitigate these, we advocate for transparency, rigorous bias testing, robust security measures, and human oversight in AI applications. Our goal is to contribute positively to the field and to encourage responsible AI development and deployment.

References

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

Anthropic. 2023. Introducing Claude.

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

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

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

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

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. 2023. Multipl-e: a scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*.

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

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, abs/2107.03374.

Hantian Ding, Varun Kumar, Yuchen Tian, Zijian Wang, Rob Kwiatkowski, Xiaopeng Li, Murali Krishna Ramanathan, Baishakhi Ray, Parminder Bhatia, Sudipta Sengupta, et al. 2023a. A static evaluation of code completion by large language models. *arXiv preprint arXiv:2306.03203*.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2023b. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. In *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*.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2022. Cocomic: Code completion by jointly modeling in-file and cross-file context. *arXiv preprint arXiv:2212.10007*.

Yifeng Ding, Hantian Ding, Shiqi Wang, Qing Sun, Varun Kumar, and Zijian Wang. 2024. Horizon-length prediction: Advancing fill-in-the-middle capabilities for code generation with lookahead planning. *arXiv preprint arXiv:2410.03103*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida I. Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, abs/2204.05999.

Linyuan Gong, Sida Wang, Mostafa Elhoushi, and Alvin Cheung. 2024. Evaluation of llms on syntax-aware code fill-in-the-middle tasks. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2024a. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Jiawei Guo, Ziming Li, Xueling Liu, Kaijing Ma, Tianyu Zheng, Zhouliang Yu, Ding Pan, Yizhi Li, Ruibo Liu, Yue Wang, et al. 2024b. Codeeditorbench: Evaluating code editing capability of large language models. *arXiv preprint arXiv:2404.03543*.

Siming Huang, Tianhao Cheng, Jason Klein Liu, Jiaran Hao, Liuyihan Song, Yang Xu, J Yang, JH Liu, Chenchen Zhang, Linzheng Chai, et al. 2024. Opencoder: The open cookbook for top-tier code large language models. *arXiv preprint arXiv:2411.04905*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Siyuan Jiang, Jia Li, He Zong, Huanyu Liu, Hao Zhu, Shukai Hu, Erlu Li, Jiazheng Ding, Yu Han, Wei Ning, Gen Wang, Yihong Dong, Kechi Zhang, and Ge Li. 2024. aixcoder-7b: A lightweight and effective large language model for code completion. *CoRR*, abs/2410.13187.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. 2023. xcodeeval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. *arXiv preprint arXiv:2303.03004*.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-Tau Yih, Daniel Fried, Sida I. Wang, and Tao Yu. 2023. DS-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 18319–18345. PMLR.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Moustafa-Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: May the source be with you! *arXiv preprint arXiv:2305.06161*, abs/2305.06161.

Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with

AlphaCode. *arXiv preprint arXiv:2203.07814*, abs/2203.07814.

Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*, pages 55–56.

Jiawei Liu, Thanh Nguyen, Mingyue Shang, Hantian Ding, Xiaopeng Li, Yu Yu, Varun Kumar, and Zijian Wang. 2024a. Learning code preference via synthetic evolution. *arXiv preprint arXiv:2410.03837*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023a. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210*, abs/2305.01210.

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

Siyao Liu, He Zhu, Jerry Liu, Shulin Xin, Aoyan Li, Rui Long, Li Chen, Jack Yang, Jinxiang Xia, ZY Peng, et al. 2024c. Fullstack bench: Evaluating llms as full stack coder. *arXiv preprint arXiv:2412.00535*.

Tianyang Liu, Canwen Xu, and Julian McAuley. 2023b. Repobench: Benchmarking repository-level code auto-completion systems. *arXiv preprint arXiv:2306.03091*.

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

Mayank Mishra, Matt Stallone, Gaoyuan Zhang, Yikang Shen, Aditya Prasad, Adriana Meza Soria, Michele Merler, Parameswaran Selvam, Saptha Surendran, Shivdeep Singh, et al. 2024. Granite code models: A family of open foundation models for code intelligence. *arXiv preprint arXiv:2405.04324*.

MistralAI. 2024. Codestral. `https://mistral.ai/news/codestral`. 2024.05.29.

Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. Codegen2: Lessons for training llms on programming and natural languages. *CoRR*, abs/2305.02309.

OpenAI. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Qiwei Peng, Yekun Chai, and Xuhong Li. 2024. Humaneval-xl: A multilingual code generation benchmark for cross-lingual natural language generalization. *arXiv preprint arXiv:2402.16694*.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. 2022. BLOOM: A 176B-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*.

Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Zhiyuan Liu, and Maosong Sun. 2024. Debugbench: Evaluating debugging capability of large language models. *arXiv preprint arXiv:2401.04621*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

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

Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. 2024. Repoformer: Selective retrieval for repository-level code completion. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

Weixiang Yan, Yuchen Tian, Yunzhe Li, Qian Chen, and Wen Wang. 2023. Codetransocean: A comprehensive multilingual benchmark for code translation. In *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*, pages 5067–5089. Association for Computational Linguistics.

Jian Yang, Jiaxi Yang, Ke Jin, Yibo Miao, Lei Zhang, Liqun Yang, Zeyu Cui, Yichang Zhang, Binyuan Hui, and Junyang Lin. 2024. Evaluating and aligning codellms on human preference. *arXiv preprint arXiv:2412.05210*.

Pengcheng Yin, Wen-Ding Li, Kefan Xiao, Abhishek Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski, et al. 2023. Natural language to code generation in interactive data science notebooks. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 126–173.

Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. Repocoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pages 2471–2484. Association for Computational Linguistics.

Heri Zhao, Jeffrey Hui, Joshua Howland, Nam Nguyen, Siqi Zuo, Andrea Hu, Christopher A. Choquette-Choo, Jingyue Shen, Joe Kelley, Kshitij Bansal, Luke Vilnis, Mateo Wirth, Paul Michel, Peter Choy, Pratik Joshi, Ravin Kumar, Sarmad Hashmi, Shubham Agrawal, Zhitao Gong, Jane Fine, Tris Warkentin, Ale Jakse Hartman, Bin Ni, Kathy Korevec, Kelly Schaefer, and Scott Huffman. 2024. Codegemma: Open code models based on gemma. *CoRR*, abs/2406.11409.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *arXiv preprint arXiv:2303.17568*, abs/2303.17568.

Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. 2024. Opencodeinterpreter: Integrating code generation with execution and refinement. *arXiv preprint arXiv:2402.14658*.

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

A Related Work (Full Version)

Code Large Language Models In software engineering, the advent of large language models (LLMs) tailored for code-centric tasks has proven to be transformative. Models (Feng et al., 2020; Chen et al., 2021; Scao et al., 2022; Li et al., 2022; Allal et al., 2023; Fried et al., 2022; Wang et al., 2021; Zheng et al., 2024; Jiang et al., 2024; Nijkamp et al., 2023; Wei et al., 2023; Zhao et al., 2024) like CodeLlama (Rozière et al., 2023), DeepSeek-Coder (Guo et al., 2024a), OpenCoder (Huang et al., 2024) and Qwen2.5-Coder (Hui et al., 2024) — all trained on vast corpuses comprising billions of code snippets — have fundamentally augmented the development process. These Code LLMs are instrumental in automating repetitive software tasks, proposing code improvements, and facilitating the conversion of natural language into executable code. Notable among these are Starcoder (Li et al., 2023; Lozhkov et al., 2024), CodeLlama (Rozière et al., 2023), and CodeQwen (Bai et al., 2023), each bringing unique contributions to the enhancement of coding assistance tools. With these advancements, Code LLMs showcase a promising trajectory for further revolutionizing how developers interact with code, promising ever-greater efficiency and intuitiveness in software creation. Inspired by the success of the grammar-based parsed tree in many fields, we adopt the abstract syntax tree to augment the code completion training.

Repo-level Code Evaluation In the domain of code evaluation, a rich tapestry of benchmarks (Zheng et al., 2023; Yu et al., 2024; Yin et al., 2023; Peng et al., 2024; Khan et al., 2023; Guo et al., 2024b; Lai et al., 2023) has been woven to address the challenges of accurately assessing code quality, functionality, and efficiency, such as HumanEval/MBPP (Chen et al., 2021; Austin et al., 2021), their upgraded version EvalPlus (Liu et al., 2023a), and the multilingual benchmark MultiPL-E (Cassano et al., 2023), McEval (Chai et al., 2024), and MdEval (Liu et al., 2024b). BigCodeBench (Zhuo et al., 2024), fullstack (Liu et al., 2024c), CodeFavor (Liu et al., 2024a), CodeArena (Yang et al., 2024) and SAFIM (Gong et al., 2024) separately evaluate code LLMs for more diverse scenarios and code preferences. The current benchmarks support code models to evaluate a series of different types of tasks, such as code understanding, code repair (Lin et al., 2017; Tian et al., 2024; Jimenez et al., 2023) and code translation (Yan et al., 2023). An important task FIM (Fried et al., 2022; Bavarian et al., 2022; Ding et al., 2024) is to fill the middle code, given the prefix and suffix code, which provides substantial assistance for software development. Repo-level completion, such as RepoEval (Zhang et al., 2023), CrossCodeEval (Ding et al., 2023b; Wu et al., 2024) and RepoBench (Liu et al., 2023b) only using exact match and edit similarity without code execution can not accurately reflect the model performance and Humaneval-Fim (Zheng et al., 2023) focus in-file evaluation.