Do Code LLMs Really Complete Repositories?

Anonymous ACL submission

Abstract

We ask whether code LLMs can truly complete real repositories, and introduce ExecRepoBench, an execution based benchmark for repository code completion. Existing benchmarks often score completions by exact match or edit similarity, but these metrics can miss the central requirement of software development: the completed code must run correctly inside the original project. ExecRepoBench contains 1.2K completion tasks from 50 active Python repositories, inserts each model prediction back into the source project, and evaluates functional correctness with unit tests across random span, line, expression, statement, and function completion. To improve repository completion, we further construct Repo-Instruct, a large instruction corpus for in file and cross file completion, and fine tune Qwen2.5-Coder into a 7B model evaluated against more than 30 open and closed LLMs. Experiments show that text similarity can misrepresent executable correctness, while our fine tuned model achieves 44.2% average Pass@1 on ExecRepoBench and remains competitive on standard coding benchmarks.[1]

1 Introduction

Code completion is one of the most widely used interfaces between developers and code LLMs. Modern code models such as DeepSeek-Coder (Guo et al., 2024a) and Qwen-Coder (Hui et al., 2024) are trained on large code corpora and can produce fluent local completions, function bodies, and program snippets. In practical software development, however, a useful completion is not merely a plausible continuation of the visible file. It must be compatible with project-wide imports, types, hidden dependencies, conventions, and tests.

Existing evaluations only partially capture this requirement. Fill-in-the-middle benchmarks derived from HumanEval or similar datasets mainly test in-file infilling with short contexts (Allal et al., 2023; Zheng et al., 2023). Repository completion benchmarks, including CrossCodeEval-style settings (Ding et al., 2023b, 2022, 2023a), incorporate cross-file context but still rely heavily on exact match, edit similarity, or other string-based metrics. These metrics are easy to compute, but they can reward completions that resemble a reference while failing to execute, and they can penalize alternative implementations that pass the same tests. This mismatch motivates a direct question: *do code LLMs really complete repositories, or do they merely reproduce reference-like text?*

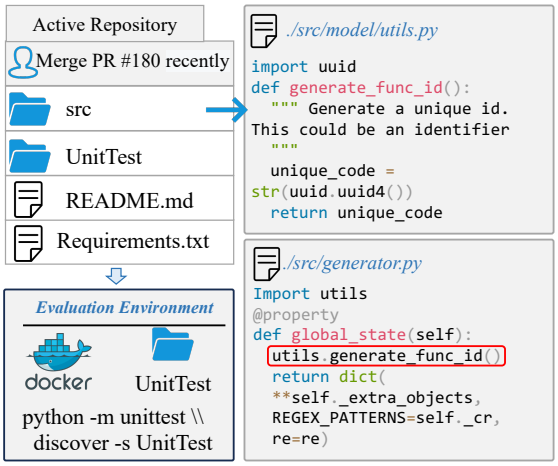


Figure 1: Execution based repository code evaluation with unit tests.

We address this question by introducing **ExecRepoBench**, an executable repository-level benchmark for code completion. As shown in Figure 1, ExecRepoBench evaluates a completion by inserting it back into the original project and running repository-level unit tests. The benchmark contains 1.2K completion tasks from 50 active Python repositories and covers diverse domains such as data processing, media and image processing, date and time utilities, and design patterns. This design turns repository completion from a text matching task into a functional correctness task.

[1]The evaluation code and dataset will be released.

ExecRepoBench also tests completions at multiple granularities. In addition to random span, single-line, and multi-line masking, we use abstract syntax trees (ASTs) to mask expressions, statements, and functions. These syntax aware tasks reflect common editing behaviors in which developers complete a condition, an assignment, a control-flow statement, or an entire function body under repository context. The same construction further supports **Repo-Instruct**, a large instruction corpus for in file and cross file completion. We use this corpus to fine tune Qwen2.5-Coder into Qwen2.5-Coder-Instruct-C, a 7B model for repository-aware completion that keeps general code generation strong.

We evaluate more than 30 open and closed LLMs on ExecRepoBench and standard code generation benchmarks (Cassano et al., 2023). The results show a clear gap between text similarity and executable correctness. They also show that Qwen2.5-Coder-Instruct-C improves executable repository completion substantially, reaching 44.2% average Pass@1 on ExecRepoBench while remaining competitive on standard code generation tasks. The contributions are summarized as follows:

- We introduce ExecRepoBench, an execution based repository code completion benchmark with 1.2K tasks from 50 active Python repositories, where completions are evaluated by running unit tests in the original projects.
- We design a multi-granularity completion protocol that covers random span and line completion as well as syntax aware expression, statement, and function completion derived from ASTs.
- We construct Repo-Instruct, a large instruction corpus for in file and cross file completion, and fine tune a 7B code LLM for executable repository completion without sacrificing code generation.

2 ExecRepoBench Construction

Data Collection and Annotation We build ExecRepoBench from active Python repositories on GitHub. Repository selection follows three criteria: the project must be actively maintained, the completion target must depend on repository context rather than only local syntax, and the repository must provide or admit unit tests that can be executed within a limited time budget. For each selected repository, we collect or write test cases and verify that the original project passes these tests before any masking is applied. This step ensures that failures after completion reflect model behavior rather than broken source repositories. Figure 2 shows that the collected repositories cover 14 domains, including 'Date and Time', 'Data Processing', 'Media&Image', 'Design Patterns', and 'Data Validation'. Each task provides the prefix and suffix of the target file together with retrieved context from the same repository, and the model predicts the missing middle code. To reduce contamination, we remove exact 20-gram word overlaps with CrossCodeEval (Ding et al., 2023b) and the Stack V2 pre-training corpus (Lozhkov et al., 2024).

Date and Time
Data Processing
Text Processing
Web Development
Code
Geospatial
Utilities
XML Parsing
Design Patterns
Raw Text
Markdown
NLP
Data Validation
Testing
Caching
Argument Parsing
Subtitles
Image Processing
Serialization
Mapping
Protocol
Algorithms&Data Structures
Chronyk
maya
moment
workdays
algorithms
boltons
transitions
html-to-json
lxml_html_clean
dict2css
objprint
edx-lint
sqlparse
attrs
PTable
geopy
fastkml
Unipath
eip712
tryme
shortuid
untangle
xmltodict
PyPattyrn
python-patterns
imageio
pysrt
easyargs
cacheout
truecase
w2n
mdformat
python-pinyin
textdistance
gruut
csv-kit
tabulator-py
csv-diff
dpath-python
eml_parser
ISO8601
Levenshtein
Markdown-it-py
Cache
Caching
CLI Tools
Image
Design Patterns
Data Parsing
Geospatial
Serialization
Testing
File Paths
Protocol
msgpack-python
Utilities more-itertools
Data Validation
Linting
Debugging
Minification
CSS
HTML Parsing
Data Structures
State Machines
Containers
Algorithms
python-sortedcontainers
Date and Time
Text Processing
Data Processing
Text Processing
Text Comparison
Text Processing
Text-to-Speech
String Similarity
Markdown
Raw Text
Caching
Argument Parsing
Subtitles
Image Processing
Design Patterns
XML Parsing
Utilities
Testing
Serialization
Mapping
Geospatial
Data Validation
Code
Web Development
Algorithms&Data Structures
Date and Time
Data Processing
Text Processing
Text Processing
Text

	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.

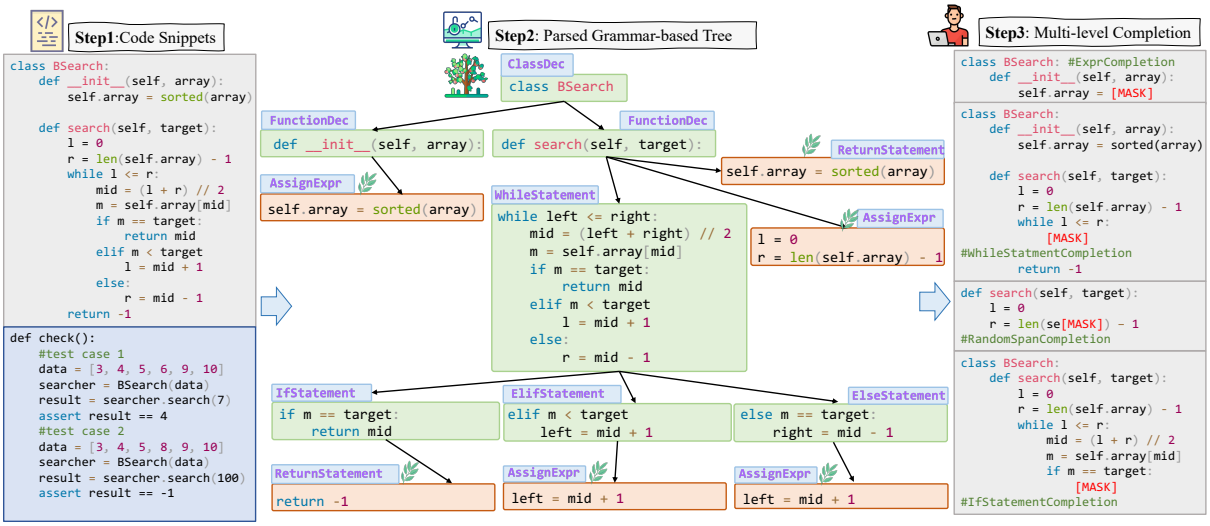


Figure 3: Multi-granularity completion based on AST parsing. We use `tree-sitter` to parse code into ASTs and extract logical units at the expression, statement, and function levels. The benchmark also includes random line and random span completion for comparison with standard fill-in-the-middle settings.

3 Multi-level Code Completion

3.1 Overview

ExecRepoBench is designed around a simple principle: a repository completion should be judged by whether the resulting repository still works. The construction pipeline therefore contains four stages. First, we select an executable repository and verify that its original tests pass. Second, we choose a target file and create a missing region either by heuristic masking or by AST-based masking. Third, we build the model prompt from the current file prefix, current file suffix, and relevant repository context. Finally, we insert the predicted code back into the original file and run the repository tests. This pipeline connects the benchmark design, the instruction data, and the final evaluation criterion under the same executable completion setting.

3.2 Problem Definition

In-file Completion Given the code x^{L_k} of a current file written in programming language L_k, where $L_k \in L_{all} = \{L_i\}_{i=1}^{K}$, the model predicts the missing middle code x_m conditioned on the prefix x_p and suffix x_s:

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

The predicted middle code is concatenated with the prefix and suffix to reconstruct the file, and the reconstructed program is evaluated with the corresponding tests.

Repository-level Completion Repository-level completion extends this setting by conditioning on code from other files in the same project. Given N context snippets $z = \{z_i^{L_k}\}_{i=1}^{N}$, the model predicts x_m from the current file prefix $x_p^{L_k}$, current file suffix $x_s^{L_k}$, and repository context c:

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

The completed file is then placed back into the original repository for execution based evaluation. Here, c denotes the concatenated repository context available to the model.

3.3 Execution based Evaluation

The key difference between ExecRepoBench and string-matching completion benchmarks is the eval-

uation target. After a model produces $\hat{x}_m$, we reconstruct the target file as $(x_p, \hat{x}_m, x_s)$ and replace the corresponding file in the original repository. We then run the repository-level unit tests used during benchmark construction. A completion is counted as correct only when the reconstructed repository passes the tests. This protocol evaluates whether the completion is compatible with the repository environment, including imports, helper functions, object states, side effects, and expected outputs.

Execution based evaluation also changes how alternative implementations are treated. If a generated completion differs from the reference but satisfies the same tests, Pass@1 credits it as correct. If a generated completion has high token overlap with the reference but breaks a dependency or violates an assertion, Pass@1 marks it as incorrect. This property is important for repository completion because real software often admits multiple valid implementations, whereas edit similarity only measures proximity to a single hidden reference.

3.4 Repository Context Construction

Repository-level completion requires deciding which code outside the target file should be visible to the model. For each masked file, we rank candidate context files by their relevance to the target file and pack selected snippets into the prompt together with the target prefix and suffix. The prompt is organized to separate repository context from the current file, so that the model can distinguish external definitions from the location that must be completed. When the combined context exceeds the supported length of the evaluated model, we truncate lower ranked context while preserving the target prefix and suffix. This design reflects practical IDE scenarios, where a completion engine must work with partial repository context and a finite context budget.

3.5 Multi-level Grammar-based Completion

Practical completion requests occur at different syntactic levels. A developer may complete an expression, a branch condition, an assignment statement, or a full function body. To model these cases, we use `tree-sitter-languages`[2] to parse each source file into an AST and select nodes at multiple logical units, as illustrated in Figure 3. The tree structure allows us to mask code fragments that are syntactically meaningful rather than arbitrary character spans.

Expression-level Completion Expression-level completion targets sub-expressions or standalone expressions, such as operands, operators, function arguments, and return expressions. This level tests whether a model can infer local semantics while respecting surrounding syntax. Because expression-level targets are short, they often have high sensitivity to variable names, imported symbols, and data types from nearby or external code.

Statement-level Completion Statement-level completion masks individual statements, including assignments and control-flow constructs. This level requires the model to preserve both syntactic validity and the local execution path of the program. A statement-level completion may determine whether an exception is raised, whether a branch condition is correct, or whether an object state is updated consistently with the rest of the repository.

Function-level Completion Function-level completion masks function bodies or function signatures. It is the most context-dependent setting because the missing code often interacts with project-level APIs, class attributes, helper functions, and tests. Function-level tasks are therefore useful for testing whether a model can synthesize coherent logic from repository context rather than only recover short local tokens.

Why multiple granularities matter The six completion types in ExecRepoBench are not interchangeable. Random span and line completion evaluate compatibility with prior fill-in-the-middle work, while AST-based completion tests units that correspond to programmer intent and language semantics. Expression tasks emphasize local binding and type constraints; statement tasks emphasize execution flow; function tasks emphasize longer-range repository behavior. Reporting these categories separately makes it possible to diagnose where a model fails instead of collapsing all completions into a single aggregate score.

3.6 Heuristic Completion Techniques

We also include two heuristic masking strategies to make the benchmark comparable with prior fill-in-the-middle evaluations.

Random Line Completion Random line completion masks individual lines in the target file. It

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

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 results on ExecRepoBench. We report ES and Pass@1 for random span, single-line, multi-line, expression, statement, and function completion.

covers short edits that are common in interactive development and provides a simple baseline for evaluating repository context.

Random Span Completion Random span completion masks contiguous lines in the target file. This setting tests whether a model can recover longer code fragments while preserving the surrounding control flow and data flow.

3.7 Hybrid Instruction Tuning

Base code models are usually exposed to fill-in-the-middle objectives during pre-training, but they are not necessarily optimized for repository-aware instruction following. Repo-Instruct is constructed to bridge this gap. Each completion example presents the repository context, the target file prefix, and the target file suffix as an instruction, and uses the masked code as the desired response. This format teaches the model to identify relevant context, infer the missing code unit, and emit only the completion rather than rewriting the whole file.

We fine tune the model with a mixture of 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 completion objective is:

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

where c denotes the repository context supplied with each completion prompt.

We combine completion data with standard instruction data $(q^{L_k}, a^{L_k}) \in D_{q,a} = \{D_{q,a}^{L_k}\}_{k=1}^K$ so that the model retains general code generation and question answering ability. This mixture is important because a pure completion model may over-specialize to infilling formats, while a pure instruction model may ignore suffix constraints and repository dependencies. The instruction tuning objective is:

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

where (q^{L_k}, a^{L_k}) denotes a query and its response from code generation, code summarization, and

other code-related instruction tasks. The hybrid objective is:

$$\mathcal{L}_{all} = \mathcal{L}_c + \mathcal{L}_{qa} \tag{5}$$

where $\mathcal{L}_c$ is the completion objective and $\mathcal{L}_{qa}$ is the instruction objective.

4 Experiments

4.1 Code LLMs

We evaluate more than 30 LLMs, covering open code models and closed general models with sizes from 0.5B to more than 30B parameters. For closed models, we evaluate GPT models (Brown et al., 2020; OpenAI, 2023), including GPT-3.5-Turbo and GPT-4o, and Claude models (Anthropic, 2023). For code models, we evaluate CodeLlama (Rozière et al., 2023), StarCoder and 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), DeepSeekCoder (Guo et al., 2024a), CodeStral (Mistral AI, 2024), Yi-Coder[3], CodeGemma (Zhao et al., 2024), and Granite-Coder (Mishra et al., 2024).

4.2 Implementation Details

We extract repository-level code snippets from `the-stack-V2`[4] and filter repositories using heuristic criteria such as GitHub stars and file length. We keep mainstream programming languages, including Python, C#, C++, Java, JavaScript, TypeScript, and PHP, and remove long-tailed languages, resulting in nearly 1.5M repositories. The resulting Repo-Instruct corpus contains nearly 3M completion samples. We fine tune Qwen2.5-Coder (Hui et al., 2024) with a mixture of this completion data and standard instruction data. Qwen2.5-Coder-Instruct-C is trained with `Megatron-LM`[5] on 64 NVIDIA H100 GPUs. The learning rate warms up to 3×10^{-4} over 100 steps and then follows a cosine decay schedule. We use the Adam optimizer (Kingma and Ba, 2015) with a global batch size of 2048 samples and truncate input sequences to 32K tokens.

[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

4.3 Evaluation Metrics

Edit Similarity We compare generated code with the reference completion using edit similarity (ES) as a string-based metric.

Pass@k We use Pass@k (Chen et al., 2021) to measure executable correctness. In this work, we report greedy Pass@1 for all models to ensure a consistent decoding setting.

4.4 Evaluation Benchmarks

Code Completion ExecRepoBench contains 1.2K completion tasks from 50 active Python repositories. Each completion is inserted into the original repository and evaluated with unit tests. We report both ES and Pass@1 to compare text similarity with executable correctness.

Code Generation Because Qwen2.5-Coder-Instruct-C is trained with both completion and instruction data, we also evaluate it on standard code generation tasks. We use MultiPL-E (Cassano et al., 2023) to test multilingual generation performance.

4.5 Main Results

ExecRepoBench Table 2 is the main evidence for executable repository completion. Qwen2.5-Coder-Instruct-C achieves the best average Pass@1 among the evaluated open models, reaching 44.2% with 7B parameters. This score is higher than substantially larger open models in the same table, including Code-Llama-70B and DS-Coder-33B, showing that repository completion is not solved by scale alone. The gain is strongest on expression and statement completion, where repository context and syntax-aware training are directly tested. These two categories require the model to recover code that is short enough to be tightly constrained by syntax, but important enough to affect execution. The table also shows that ES and Pass@1 do not always agree: Granite-Coder-8B has low ES but competitive Pass@1, indicating that text similarity can understate correct alternatives. This mismatch supports the central claim of ExecRepoBench: repository completion should be evaluated by execution when tests are available.

Code Generation Table 3 verifies that repository completion tuning does not collapse general code generation. Qwen2.5-Coder-Instruct-C reaches 76.4% average Pass@1, matching the 7B-scale Qwen2.5-Coder baseline while adding stronger executable completion performance. This result

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

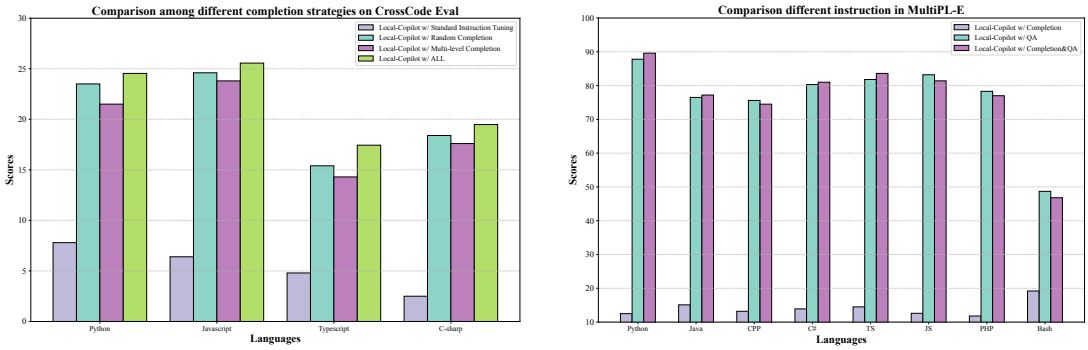
Table 3: The performance of different instruction LLMs on EvalPlus and MultiPL-E. "HE" denotes the HumanEval, "HE+" denotes the plus version with more test cases, and "MBPP+" denotes the plus version with more test cases.

is important because a model optimized only for repository completion could learn to emit narrow infilling responses and lose broader programming ability. The hybrid tuning recipe avoids this failure mode: completion data teaches prefix, suffix, and repository context use, while instruction data preserves standard code generation behavior across EvalPlus and MultiPL-E.

5 Analysis

Execution exposes metric mismatch. Table 2 shows why ES alone is insufficient. ES measures proximity to one reference, while Pass@1 checks whether the patched repository passes tests. The Granite-Coder-8B row is a clear example: its ES is low, but its Pass@1 is competitive with much larger models. Because each generated completion is inserted back into the repository before testing, ExecRepoBench exposes failures caused by wrong imports, violated object states, or plausible code that does not satisfy project tests.

Granularity localizes model gains. The multi-granularity design makes the improvement interpretable. In Table 2, Qwen2.5-Coder-Instruct-C reaches 58.7% Pass@1 on expression completion and 45.5% on statement completion, the strongest gains among the listed task types. Function completion remains harder, which is consistent with its longer dependencies over helper functions, class state, and project-specific conventions. Reporting



(a) CrossCodeEval (b) MultiPl-E

Figure 4: Ablation results on code completion and code generation. Combining standard instruction data with completion data helps Qwen2.5-Coder-Instruct-C support both repository completion and general code generation.

these categories separately prevents an aggregate score from hiding where repository completion still fails.

Instruction mixture preserves generation. Figure 4 and Table 3 support the hybrid training design. Completion data improves repository completion, while standard instruction data keeps code generation competitive. This evidence justifies training Qwen2.5-Coder-Instruct-C with both Repo-Instruct and standard instruction samples.

Ablation Study Figure 4 compares completion-only, instruction-only, and mixed training. The mixed setting gives the strongest combined behavior across CrossCodeEval and MultiPL-E, which matches the goal of building one model for repository completion and standard code generation. Completion examples align the model with prefix, suffix, and repository context, while standard instruction examples keep the model useful for standalone programming tasks.

6 Related Work

Code Large Language Models Code LLMs support generation, repair, explanation, and completion (Feng et al., 2020; Chen et al., 2021; Li et al., 2022; Allal et al., 2023; Fried et al., 2022; Wang et al., 2021; Zhao et al., 2024; Rozière et al., 2023; Guo et al., 2024a; Huang et al., 2024; Hui et al., 2024). Many are trained with fill-in-the-middle objectives, but their behavior in executable repository settings remains less directly measured.

Repo-level Code Evaluation Code evaluation has moved from standalone problems (Chen et al., 2021; Austin et al., 2021; Liu et al., 2023a) to broader software tasks (Zhuo et al., 2024; Yang et al., 2024; Gong et al., 2024; Zheng et al., 2023; Yu et al., 2024; Yin et al., 2023; Lai et al., 2023; Cheng et al., 2024; Liu et al., 2024a). Repository completion benchmarks such as RepoCoder (Zhang et al., 2023), CrossCodeEval (Ding et al., 2023b; Wu et al., 2024), and RepoBench (Liu et al., 2023b) introduce cross-file context, but mainly rely on EM or ES. ExecRepoBench complements them by evaluating whether completed code executes correctly in the original repository.

7 Conclusion

We introduced ExecRepoBench, an execution based benchmark for repository code completion, and Repo-Instruct, an instruction corpus for in file and cross file completion. ExecRepoBench inserts completions into real Python repositories and runs unit tests, distinguishing executable code from reference-like text. Its multi-granularity design evaluates random spans, lines, expressions, statements, and functions separately, making repository completion failures easier to localize. Results across more than 30 LLMs show that ES can diverge from Pass@1, and that Qwen2.5-Coder-Instruct-C improves executable completion while preserving competitive code generation. More broadly, the benchmark and training recipe form a closed loop: repository-aware prompts are trained with prefix, suffix, and cross-file context, then judged by project execution. This makes the gains actionable for real development settings, where completions must satisfy syntax, dependencies, and tests together. Repository completion should therefore be evaluated as a software execution problem, not only as string matching.

Limitations

We identify three limitations. First, ExecRepoBench focuses on Python repositories, and executable multilingual repository completion remains an important direction for future work. Second, Qwen2.5-Coder-Instruct-C is fine tuned from a 7B base model; scaling the same training recipe to larger and smaller models may reveal different tradeoffs between completion and generation. Third, the current model is trained with supervised fine tuning only. Preference optimization methods such as DPO may further improve usability and response formatting in interactive development settings.

Ethics Statement

This work studies code completion in open-source repositories and is intended to improve the evaluation of AI-assisted programming systems. Potential risks include reproducing insecure code patterns, generating incorrect completions that pass incomplete tests, and exposing models to licensing or privacy issues in large code corpora. We mitigate these risks by evaluating functional correctness with tests, documenting benchmark construction, and planning to release the evaluation code and dataset for inspection. Users should still review generated code before deployment, especially in security-sensitive software.

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, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2023. MultiPL-E: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7):3675–3691.

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.

Yao Cheng, Jianfeng Chen, Jie Chen, Li Chen, Liyu Chen, Wentao Chen, Zhengyu Chen, Shijie Geng, Aoyan Li, Bo Li, et al. 2024. FullStack Bench: Evaluating LLMs as full stack coders. *arXiv preprint arXiv:2412.00535*.

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. Planning-aware code infilling via horizon-length prediction. *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 processing. *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*.

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*.

Mistral AI. 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, Romain Sauvestre, Tal Remez, 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, Yinxu Pan, Yesai Wu, Haotian Hui, Weichuan Liu, 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: Empowering code generation with OSS-Instruct. *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, Oleksandr Polozov, and Charles Sutton. 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 benchmarking 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 (Cheng et al., 2024), 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 RepoCoder (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.