Can Language Models Replace Programmers for Coding?

REPOCOD Says 'Not Yet' *

Shanchao Liang
Purdue University
liang422@purdue.edu

Nan Jiang
Purdue University
jiang719@purdue.edu

Yiran Hu
Purdue University
hu954@purdue.edu

Lin Tan
Purdue University
lintan@purdue.edu

Abstract

Recently, a number of repository-level code generation benchmarks—such as CoderEval, DevEval, RepoEval, RepoBench, and LongCode-Arena—have emerged to evaluate the capabilities of large language models (LLMs) beyond standalone benchmarks like HumanEval and MBPP. Thus, a natural question is, would LLMs have similar performance in real world coding tasks as their performance in these benchmarks? Unfortunately, one cannot answer this question, since these benchmarks consist of short completions, synthetic examples, or focus on limited scale repositories, failing to represent real-world coding tasks.

To address these challenges, we create REPOCOD, a Python code-generation benchmark containing complex tasks with realistic dependencies in real-world large projects and appropriate metrics for evaluating source code. It includes 980 whole-function generation tasks from 11 popular projects, 50.8 % of which require repository-level context. REPOCOD includes 314 developer-written test cases per instance for better evaluation. We evaluate ten LLMs on REPOCOD and find that none achieves more than 30% pass@1 on REPOCOD, indicating the necessity of building stronger LLMs that can help developers in real-world software development. In addition, we found that retrieval-augmented generation achieves better results than using target function dependencies as context.

1 Introduction

One critical application of large language models (Bian et al., 2023; Zheng et al., 2023) is code generation (Li et al., 2023a; Ouyang et al., 2023), where models generate executable code snippets given natural language descriptions (Li et al., 2023b; Lozhkov et al., 2024; Touvron et al., 2023; Dubey et al., 2024; Achiam et al., 2023). Such code generation tasks are an integral part of software development.

Research of LLMs for code generation requires a high-quality dataset that evaluates LLMs' ability to code in real-world development scenarios. Existing LLMs achieve high accuracies, i.e., over 90% pass@1, in solving Python coding tasks in self-contained benchmarks. Would these LLMs achieve similar accuracies in real-world code-generation tasks? To answer this question, we need benchmarks that represent real-world code generation tasks. Specifically, benchmarks need to meet the following four criteria:

Firstly, tasks should be **real-world coding tasks** such as implementing complete functions based on specifications. While manually-crafted benchmarks such as HumanEval and MBPP are useful at the earlier development of LLMs for code, they fail to represent realistic software development, where real project requirements drive tasks.

Secondly, **both tasks and source repositories need to be complex**, as prior work (Kiela et al., 2021; Ott et al., 2022) highlights that benchmarks quickly become saturated as models evolve and outperform them. To evaluate the true capabilities of current and future LLMs, it is crucial to construct benchmarks with challenging tasks (Jimenez et al., 2024; bench authors, 2023). Benchmarks such as CoderEval (Zhang et al., 2024), DevEval (Li et al., 2024b), RepoEval (Zhang et al., 2023), RepoBench (Liu et al., 2024), and Long-Code-Arena (Bogomolov et al., 2024), focus on single-line completion or short-function generation, failing to capture the complexity of multi-hundred-line functions commonly found in real-world projects. In addition, the repositories need to be complex to evaluate LLMs' ability to understand and navigate complex context. However, existing code generation benchmarks overlook this aspect, with the average number of files in their source repositories

*To appear in the Proceedings of ACL 2025. This version is for personal use only.

fewer than 200. Without a stronger benchmark that incorporates complex tasks and large-scale projects, we cannot reliably assess the boundaries of LLMs' code generation capabilities.

Thirdly, tasks should have **repository-level dependencies.** Real-world code rarely exists in isolation—only 27% of functions in 500 repositories are *self-contained*, meaning they do not invoke other functions or modules within the repository and only use standard or public libraries (Li et al., 2024b). Most functions in real-world repositories depend on other components in the repository. Benchmarks such as CoderEval (Zhang et al., 2024) and DevEval (Li et al., 2024b) have the majority of tasks as self-contained or with simple dependencies, such as those dependent on the current file only. Only 10–31% of their tasks require dependencies beyond their current file, failing to represent coding tasks that require repository-level dependencies.

Finally, tasks should be evaluated with **appropriate evaluation metrics.** Similarity- or matching-based evaluation metrics, such as CodeBLEU (Ren et al., 2020) and BLEU (Papineni et al., 2002) measure textual similarities, and fail to determine whether two code snippets are functionally equivalent. Previous work has shown CodeBLEU and BLEU are unreliable metrics for code generation due to their high mismatch rate with human assessment (Evtikhiev et al., 2023). Execution-based evaluation (e.g., unit tests), although imperfect, avoids these pitfalls by directly verifying functionality. Appendix A.2 uses examples to show how CodeBLEU makes mistakes.

This paper designs and builds REPOCOD, a benchmark for assessing LLMs' ability in real-world coding tasks, i.e., generating *complex* functions that require *repository-level dependencies* in real-world projects, using developer *test cases* as the validation method.

This paper makes the following contributions:

- A dataset of 980 challenging coding tasks from 11 popular Python projects, characterized by:
  - Complex canonical solutions (Avg. 331.6 tokens per instance).
  - Extensive context from large repositories (Avg. 2,610 files per repository).
  - A focus on repository-level tasks—over half (498, 50.8 %) of tasks require context from other files in the repository.
  - Rigorous evaluation (Avg. 314 developer-written test cases per instance).
- A novel test collection pipeline that reduces the evaluation time to 10.4% of the original
- A thorough study of ten LLMs' performance on REPOCOD, and the key findings include:
  - LLMs are ineffective on REPOCOD (only 28.6% pass@1 under oracle-retrieval).
  - Higher recall in retrieving a target function's dependencies as context improves LLMs' performance.
  - Providing dependencies is suboptimal for repository-level code generation tasks.
  - For self-contained functions, additional context can still help generate better solutions.
  - LLMs exhibit diverse strengths, as each model has uniquely solved tasks.

2 REPOCOD Benchmark

This section introduces REPOCOD's automated data collection process (Section 2.1), data structure (Section 2.2) , and the statistics (Section 2.3).

2.1 Data Collection Pipeline

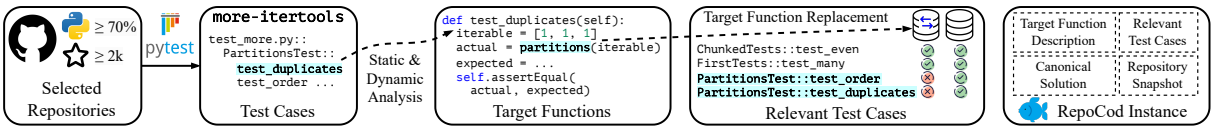
Figure 1 (a) shows the data collection pipeline. REPOCOD utilizes GitHub as the source of our data. To filter noisy data such as functions with missing descriptions and test cases, we employ a three-stage data collection pipeline to efficiently collect target functions with good documentation and the corresponding test cases for validation.

Step I: Repository Selection. The selection criteria include open-sourced repositories where Python is the primary language ($\geq$70%) and those with no less than 2k stars, as popular repositories tend to be well-maintained (Jimenez et al., 2024). We clone the latest version (as of October 2024) of these repositories for analysis in *Step II* and *Step III*.

Step II: Target Function Selection. For each collected repository, we perform both static and dynamic analysis to accurately identify the functions defined in the repository that are invoked by the developer-provided test cases, with detailed docstrings as function specifications.

We first collect all the test functions in a repository using PyTest's test discovery rules[1]. Then, we identify the functions invoked within these test functions using static and dynamic analysis techniques. For static analysis, we use tree-sitter (Brunsfeld et al., 2024) to parse the test functions and collect the functions that are invoked. For

[1] https://docs.pytest.org/en/stable/announce/release-8.3.0.html



(a): Benchmark Data Collection Pipeline (b): Benchmark Structure

Figure 1: Data collection pipeline and instance structure of REPOCOD.

dynamic analysis, Python's trace module is used to examine the execution of each test case. This approach also identifies the indirect function calls invoked by test functions, which are challenging to detect through static analysis alone.

Finally, we filter the target functions, retaining only those with more than ten lines of docstrings and at least two lines of function body.

Step III: Relevant Test Cases Collection. To validate the correctness of LLM-generated code efficiently, we avoid running the entire test suite and instead execute only the relevant test cases—those that specifically invoke the target functions. This targeted approach significantly reduces verification time in large repositories with extensive test suites.

While the previous step provides a mapping between target functions and relevant test cases, certain mappings may still be missed. For instance, test functions might use `subprocess.run(command)` to invoke repository functions indirectly, which is not collected by static or simple dynamic analysis. Therefore, we employ a two-step collection procedure to capture all relevant test cases. First, we execute all test cases in the repository to establish a reference result. Then, for each target function, we replace it with an assertion failure, rerun all test cases, and compare the results to the reference. If a test result changes from *pass* to *fail*, it indicates that the test case is relevant to the target function.

2.2 Benchmark Structure

Figure 1 (b) illustrates the components of REPOCOD's instances: the *target function description*, *repository snapshot*, *relevant test cases*, and *canonical solution*. REPOCOD uses the developer-provided docstring as the *target function description*. The *repository snapshot*, is the source repository with the target function body removed.

To use REPOCOD as a code generation benchmark, the LLM is provided with the target function description and the repository snapshot, then it is expected to generate a functionally equivalent code snippet as the canonical solution. The LLM-generated solution is considered correct if it passes all *relevant tests*. Finally, the *canonical solutions* are the developer-written function bodies.

Benchmarks	Instance Statistics				Repository Scale	
	#Instances	#Tokens	Cyclo.	#Tests	#Lines	#Files
CrossCodeEval	2,665	13.2	1.0	0	-	-
RepoBench	23,561	13.7	1.0	0	-	-
LCA	934	12.0	1.0	0	-	-
CoderEval[1]	230	108.2	4.7	-	48,821	152
DevEval[2]	1,825	86.3	3.5	2.1	36,640	164
R2E-Eval1[3]	246	127.2	-	11.5	-	-
RepoEval*	373	84.1	2.7	2742.5	7,387	119
REPOCOD	980	331.6	9.0	313.5	290,110	2,610

RepoEval*'s #Tests is averaged over a subset of executable repositories, as some repositories cannot be run to collect test case counts.

Table 1: Benchmarks Comparison. '-' indicates the data is not publicly available. **#Instances**: number of instances in each benchmark; **#Tokens**: average number of tokens of the canonical solution; **Cyclo.** average cyclomatic complexity of the canonical solution; **#Tests**: average number of test cases per instance; **#Lines**: average number of lines per repositories;

2.3 Benchmark Statistics

Table 1 compares REPOCOD with existing code generation benchmarks that use repository-level context, including CrossCodeEval (Ding et al., 2023), RepoBench (Liu et al., 2024), Long-Code-Arena (LCA) (Bogomolov et al., 2024), CoderEval (Zhang et al., 2024), DevEval (Li et al., 2024b), R2E-Eval1 (Jain et al., 2024), and RepoEval (Zhang et al., 2023).

CrossCodeEval, RepoBench, and LCA contain only single-line code generation tasks, resulting in shorter canonical solutions compared to REPOCOD. In addition, they rely on similarity-based metrics, which are insufficient to evaluate code quality. Therefore, we mainly compare REPOCOD against datasets that require full function generation and include test cases for evaluation, such as CoderEval, RepoEval and etc. Among all function generation datasets, REPOCOD outperforms in

[1]CoderEval only releases the task instances along with ground truths but omits the test cases.

[2]For data statistics not available in DevEval's paper, we recompute the data from their released dataset available in their GitHub repository.

[3]R2E-Eval1 has not released its dataset, and the data is derived from the published paper.

Dataset	Repository-Level	File-Level	Self-Contained
CoderEval	10.0	53.5	36.5
DevEval	31.3	41.2	27.5
REPOCOD	**50.8**	18.1	31.1

Table 2: Data context complexity distribution (%) of code generation benchmarks.

nearly every aspect, with the exception of having fewer instances compared to DevEval. However, REPOCOD leverages an automated annotation process for annotating the task descriptions for each instance, whereas DevEval relies on human effort, making REPOCOD more scalable for expansion.

Repository Scale. The instances in REPOCOD are collected from repositories with an average of 2,610 files and 290,110 lines of code per repository. This significantly exceeds the scale of source repositories from other benchmarks, underscoring the challenge of efficiently utilizing repository-level context for the evaluated models.

Test Scale. REPOCOD utilizes an average of 313.5 tests per instance, significantly surpassing other benchmarks, highlighting REPOCOD' robustness in evaluating models. RepoEval*'s #Tests is computed as the average number of test cases across all available repositories, regardless of the tests' relevance to the task. If measured the same way, REPOCOD' average #Test would be 17,974.

Length Complexity. REPOCOD presents the most challenging tasks among all datasets. With the largest average token count for canonical solutions at 331.6 (Table 1), LLMs must generate solutions that may require more than twice as much code as those in other benchmarks to complete the task.

Cyclomatic Complexity. REPOCOD also includes a cyclomatic complexity (McCabe, 1976) score of 9.0, indicating that the canonical solutions in REPOCOD have a higher structural complexity regarding control flow.

Context Complexity. Table 2 shows REPOCOD, DevEval and CoderEval's distribution of tasks by three types of context complexity: *Self-Contained*, *File-Level*, and *Repository-Level*. Self-Contained functions use only standard or public libraries; File-Level functions reference functions or classes from the same file but not others; Repository-Level functions may invoke functions or classes from the same file or other files in the repository. Among these benchmarks, REPOCOD has the highest ratio of repository-level dependencies (498).

These statistics show that the tasks in REPOCOD are the most challenging, and contain more repository-level context than other benchmarks, making it particularly suitable for evaluating current and future models on repository-level tasks.

Optimized Test Execution. With our test-collection pipeline, we reduce the number of required test cases per instance from an average of 17,974 (all available test cases from the source repository) to just 313, cutting execution time from 216.9 hours to 22.6 hours, demonstrating a substantial improvement in efficiency.

3 Experiment Setup

Due to the large repository sizes, most LLMs face context window limitations that prevent them from processing all the context in REPOCOD. To address this, we employ Retrieval-Augmented Generation (RAG). We evaluate three popular retrieval settings: RAG_{BM25}, RAG_{Dense}, and *current file*. In addition to a *Baseline* (no context) setting, we use two unrealistic "oracle" settings, *Callees*, and $RAG_{Dense\text{-}oracle}$, to study the potential best scenarios.

3.1 Retrieval Settings

We first define our retrieval corpus, which is used across all RAG methods. The retrieval corpus contains complete function definitions, each including the function signature and full body, sourced from all functions in the repositories.For RAG_{BM25} and RAG_{Dense}, since the target function body is unavailable during inference, we query using its signature and docstring to retrieve similar functions.

RAG_{BM25}. We use BM25 (Robertson and Zaragoza, 2009) to extract relevant functions from the corpus as context for generation. The signatures, docstrings, file paths, and bodies of the retrieved functions are provided as context.

RAG_{Dense}. We encode all functions in the corpus using the `text-embedding-3-small` model. The target function's signature and docstring are encoded into an embedding to be compared against the function embeddings in the corpus. The top-ranked functions based on cosine similarity are provided as context, in the same format as sparse retrieval.

Current File. In this setting, the context is limited to the file containing the target function, with the entire file provided as context, excluding the target function's body.

3.2 Additional Settings

Baseline. In this setting, the models are provided only with the function signature and the docstring to generate the target function. This serves as a minimal context setting, evaluating the model's ability to generate functions without context information.

Callees. Functions invoked by the canonical solution are considered as oracle context in several benchmarks (Zhang et al., 2024; Li et al., 2024b). We extract these invoked functions and include them along with the last 1,024 tokens from the target function's prefix. This setup evaluates whether explicitly providing referenced functions, with the target function's most recent context, improves generation quality.

RAG$_{\textit{Dense-oracle}}$. We include the canonical solutions along with the task descriptions as queries to perform RAG$_{\text{Dense}}$. This setting establishes an upper bound on retrieval effectiveness for RAG$_{\text{Dense}}$, showing the potential of an ideal retrieval system.

The Callees and RAG$_{\text{Dense-oracle}}$ settings are unrealistic in practice, as developers cannot obtain the exact function dependencies beforehand nor have access to the correct solutions.

3.3 Task Formulation

Each data instance of REPOCOD consists of a repository snapshot, target function signature, and docstring (from Section 2.2). The task is to generate the target function that passes all relevant test cases. In practice, LLMs are provided with a prompt consisting of the retrieved context and asked to generate the solution.

Once the model generates the function, the synthesized code is inserted back into the repository to perform test execution. The solution is considered to pass if all relevant test cases pass. We provide the details of prompt construction in Appendix A.3.

We report the models' performance using Pass@1 (Chen et al., 2021) that measures the probability that the first generated code sample is correct.

3.4 Model Setup

Given the complexity of REPOCOD, we evaluate only LLMs that meet the following criteria: state-of-the-art (SOTA) performance on existing benchmarks, and a context length of at least 16K.

Thus, we evaluate GPT-4o, GPT-4o-mini, DeepSeek-V2.5, and Claude 3.5 Sonnet, representing commercial LLMs. We also evaluate open-source LLMs such as CodeLlama, DeepSeek-Coder, and OpenCodeInterpreter series, ranging from 6.7B to 34B parameters. For commercial models, we use their official API, and for the open-source LLMs, we use the implementations provided by HuggingFace and vLLM (Kwon et al., 2023). Under each experimental setting (retrieval approach), we let each LLM generate one output per instance in REPOCOD using greedy decoding.

Models	RAG$_{\text{BM25}}$	RAG$_{\text{Dense}}$	Current-File
CodeLlama-7B	10.7	10.4	5.7
CodeLlama-34B	12.4	12.8	9.6
DeepSeekCoder-6.7B	14.0	14.1	10.9
DeepSeekCoder-33B	16.7	17.1	14.9
OpenCodeInterpreter-6.7B	12.1	12.5	13.2
OpenCodeInterpreter-33B	15.3	16.3	18.3
Claude 3.5 Sonnet	14.4	17.5	19.8
DeepSeek-V2.5	18.5	20.7	27.0
GPT-4o-Mini	15.1	15.0	18.7
GPT-4o	27.4	27.0	26.8

Table 3: Pass@1(%) of SOTA LLMs on REPOCOD.

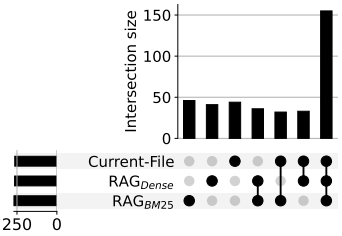


Figure 2: GPT-4o's passed tasks in different retrievals.

4 Result

4.1 SOTA LLMs' Pass@1 on REPOCOD

Table 3 shows ten LLMs' performance on REPOCOD, under three retrieval approaches. On all retrieval approaches, commercial LLMs perform better. Specifically, GPT-4o has the best result, reaching up to 27.4% pass@1. On the other hand, none of the open-sourced LLMs has over 20% pass@1.

This result shows that SOTA LLMs still struggle with repository-level code generation. Compared to their pass@1 on HumanEval (about 90% (Guo et al., 2024)), SOTA LLMs are still **far away from writing real-world programs requiring repository-level information**. We provide a detailed discussion of the results below.

Impact of Model Size. Table 3 demonstrates the consistent advantage of larger models in solving complex repository-level tasks compared to smaller models within the same architecture.

Impact of Retrieval Approach. The pass@1 results are higher for commercial models when using contexts from the current-file setting, while

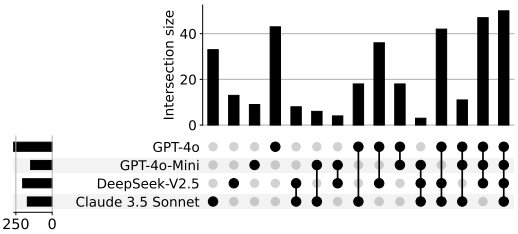


Figure 3: Commercial LLMs' passed tasks.

Model	Method	Recall Range		
		0	(0, 0.50]	(0.50, 1]
DeepSeek-v2.5	RAG_{BM25}	10.4	10.9	28.8
	RAG_{Dense}	12.3	11.0	27.2
Claude-3.5	RAG_{BM25}	7.1	4.5	24.7
	RAG_{Dense}	7.2	8.7	28.3
GPT-4o	RAG_{BM25}	20.3	16.4	41.1
	RAG_{Dense}	16.7	14.2	38.0
GPT-4o-Mini	RAG_{BM25}	10.8	7.3	24.7
	RAG_{Dense}	7.7	9.4	28.3

Table 4: Pass@1 by Recall for RAG_{BM25} and RAG_{Dense}. Distribution: (BM25: 492/110/73, Dense: 456/127/92).

RAG_{BM25} and RAG_{Dense} yield similar but lower outcomes. This trend is also observed in the OpenCodeInterpreter series of LLMs. In contrast, other open-source LLMs perform better with RAG_{BM25} and RAG_{Dense} compared to the current-file setting. These results suggest that commercial models are more likely to benefit from contexts drawn from the current-file setting.

To further investigate the overlap and uniqueness of tasks solved using different retrieval approaches, Figure 2 is an UpSet plot (Lex et al., 2014) that illustrates the overlap and uniqueness of model results using GPT-4o's result. Though most solutions are solvable by all approaches, each approach has uniquely solved tasks. This suggests that retrieval approaches differ in overall effectiveness and may capture complementary contextual information.

Model-Specific Solution Diversity. Beyond retrieval approaches, we also analyze the distribution of uniquely solved tasks by each model using the same context. Figure 3 shows the number of passed tasks for each commercial LLM and their intersections in REPOCOD under the RAG_{Dense}. Notably, each model solves a unique set of tasks, highlighting the specialized capabilities of individual models. Particularly, Claude 3.5 Sonnet and GPT-4o each have a high number of uniquely solved tasks, indicating that they each excel at certain tasks.

4.2 Impact of Complexity on Performance

We show the performance of SOTA LLMs negatively correlates with the level of complexities. We compare three types of complexities: *Context Complexity*, *Canonical Solution Length*, and *Cyclomatic Complexity levels*. We demonstrate the result in the RAG_{Dense} setting.

Context Complexity. Figure 4a details the comparison of the evaluation results of SOTA LLMs for tasks with different context complexities. All models have the lowest pass@1 when generating functions with repository-level context compared to functions with less complex dependencies. The overall pass rate decreases as the complexity of the context level increases.

Cyclomatic Complexity. Figure 4b shows that LLM performance declines as cyclomatic complexity increases. In the most challenging setting, the best LLM achieves only a 7.0% pass@1 rate.

Token Length. Figure 4b presents the performance of LLMs on tasks that require generating functions of varying token lengths. The pass@1 of the LLMs gradually decreases as the length complexity of the functions increases. In the more challenging scenarios (length > 232), even the top-performing model, GPT-4o, achieves less than 10% pass@1.

With the highest level of complexities in all three categories, REPOCOD establishes a new standard for LLMs in code generation.

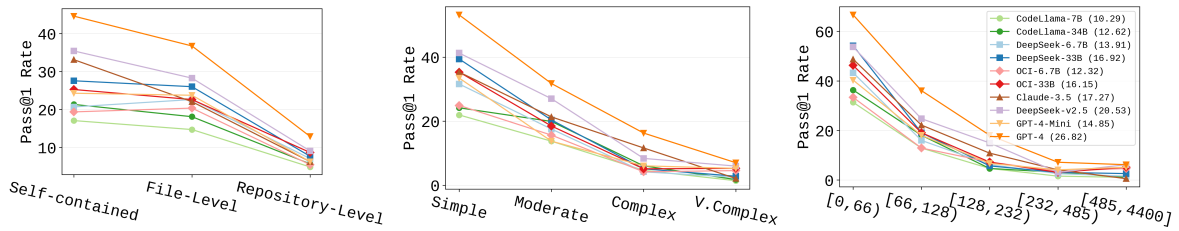
4.3 Impact of Recall on Performance

We study the impact of retrieval recall (recall of target function dependencies) on LLMs' pass rates in RAG_{BM25} and RAG_{Dense}. Table 4 presents the pass@1 performance of LLMs for retrieved content across different recall rate ranges: 0, (0, 0.5], and (0.5, 1]. Interestingly, When recall falls within (0, 0.5], model performance is comparable to that with a 0 recall rate. In contrast, **high recall (in range (0.5,1]) in retrieved content consistently leads to better model performance,** indicating the importance of retrieving target function dependencies.

4.4 Impact of Context Type on Performance

We compare the three retrieval approaches with three additional settings using GPT-4o and GPT-4o-mini in Table 5.

Finding 1: SOTA LLMs underperform on REPOCOD even with $RAG_{Dense\text{-}oracle}$. Despite using canonical solutions as queries, $RAG_{Dense\text{-}oracle}$ only marginally outperforms RAG_{Dense}. Further analysis reveals that, on average, 22.4 of the top 30 retrieved functions **(75%)** overlap between



(a) Pass@1(%) of LLMs across different context complexities.

(b) Pass@1(%) of LLMs across different cyclomatic complexities.

(c) Pass@1(%) of LLMs across different length complexities.

Figure 4: Pass@1 under context complexity and length complexity. Let n represent the number of instances in each bin. Length complexities distribution:: **[0, 66)**: n=201, **[66, 128)** : n=194, **[128, 232)**: n=194, **[232, 435)**: n=196, **[435, 4308]**: n=195. Cyclomatic complexity distribution (M): **Simple** ($M \leq 2$): n=269, **Moderate** ($3 \leq M \leq 5$): n=211 , **Complex** ($6 \leq M \leq 10$): n=215, and **Very Complex** ($M \geq 11$): n=285.

Model	Method	Self	File	Repo.	Overall
GPT-4o-Mini	Baseline	11.5	7.3	2.6	6.2
	RAG_{BM25}	22.6	22.0	8.0	15.1
	$\text{RAG}_{\text{Dense}}$	24.3	23.7	6.2	15.0
	Current-File	30.5	23.7	9.6	18.7
	Callees	27.9	23.7	7.0	16.5
	$\text{RAG}_{\text{Dense-oracle}}$	28.2	24.3	5.2	15.8
GPT-4o	Baseline	23.6	11.3	3.8	11.3
	RAG_{BM25}	39.3	31.1	18.7	27.3
	$\text{RAG}_{\text{Dense}}$	44.6	36.7	12.9	27.0
	Current-File	39.3	35.0	16.3	26.8
	Callees	35.1	31.1	12.2	22.8
	$\text{RAG}_{\text{Dense-oracle}}$	45.2	34.5	16.3	28.6

Table 5: Pass@1 (%) of GPT-4o across context complexities. Self, File, and Repo. refer to Self-Contained, File-Level and Repository-Level.

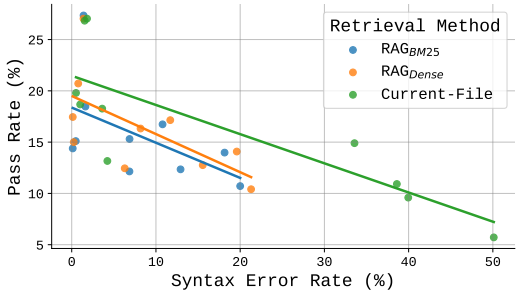


Figure 5: Models' pass@1 against syntax error rate in different retrievals. Each pair of model and retrieval approach is represented as a dot.

$\text{RAG}_{\text{Dense}}$ and $\text{RAG}_{\text{Dense-oracle}}$, suggesting that task descriptions alone enable effective retrieval in REPOCOD. This result underscores that even with an ideal retrieval setup, SOTA LLMs struggle on REPOCOD tasks, highlighting the limitations of existing LLMs for real-world coding tasks.

Finding 2: Additional context significantly improves performance for all types of tasks in REPOCOD. All settings are better than the Baseline setting. Even for self-contained tasks, the inclusion of 1,024 prefix tokens (Callees setting) improves the pass@1 result, suggesting that additional context—beyond just the task description—improves generation performance.

Finding 3: Dependency-based context is suboptimal. The Callees setting is comparable to RAG_{BM25} and $\text{RAG}_{\text{Dense}}$ but does not consistently outperform them. Across all complexity levels, it lags behind the Current-File setting. This indicates that solely using dependencies is not ideal for repository-level code generation. Combined with the findings from Section 4.3, future work should aim for retrieving contents with a high recall of the target function's dependencies while maintaining high similarity to the task description.

4.5 Failure Reasons

We compute the ratio of syntax errors as the proportion of tasks with syntax errors among all failed samples and present the results of all combinations of models (10 models) and retrieval approaches in Figure 5. Results show that the Current-File setting leads to a higher syntax error ratio, and models with a higher syntax error ratio tend to achieve lower Pass@1 performance. For RAG_{BM25} and $\text{RAG}_{\text{Dense}}$ methods, the syntax error ratio remains below 25%, suggesting that LLMs are less likely to produce syntax errors when leveraging these retrieval strategies.

We study GPT-4o's failures on 30 randomly sampled tasks and find two primary failure reasons. First, LLM-generated functions frequently lack sufficient input validation and parameter handling, such as ignoring or improperly transforming input values, leading to errors when processing unexpected inputs or misusing function arguments. Second, incorrect or incomplete implementation of core logic leads to deviations from the intended functionality, including selecting suboptimal computational methods, mismanaging object states, or misapplying parallel processing strategies. We provide case studies in the Appendix A.7.1.

5 Related Work

5.1 LLMs for Code

LLMs have been widely used for code-related tasks, among which code generation is the most important and common task (Achiam et al., 2023; Li et al., 2023b; Lozhkov et al., 2024; Nijkamp et al., 2023; Rozière et al., 2024; Guo et al., 2024; Luo et al., 2024). Existing work has shown an impressive ability to generate self-contained programs.

However, the performance of LLMs for code generation within the context of repositories remains underexplored. Despite the use of retrieval-augmented generation (RAG) to retrieve relevant context from the repository (Ding et al., 2023; Zhang et al., 2024; Wu et al., 2024a), generating code within the context of the repositories remains more challenging and yields lower accuracy compared to generating self-contained code.

5.2 Code Generation Benchmarks

Code generation benchmarks are important for evaluating LLMs' ability to generate code. Self-contained benchmarks (Hendrycks et al., 2021; Li et al., 2022; Xia et al., 2024; Liu et al., 2023) are unable to evaluate LLMs' capability to generate code for real-world projects.

A few repository-level code generation benchmarks have been developed. Concode (Iyer et al., 2018) represents an early approach in this domain, focusing on repository-level code generation evaluated through similarity-based metrics rather than the test-based metrics used in REPOCOD. CrossCodeEval (Ding et al., 2023), RepoBench (Liu et al., 2024), and Long-Code-Arena (Bogomolov et al., 2024) are benchmarks collected from GitHub projects. They consist of tasks generating a single line of code given the incomplete code snippets and repository context. As discussed in the introduction, they share two limitations: (1) their target code is limited to single-line outputs, and (2) they rely on similarity-based evaluation metrics.

Other benchmarks, such as CoderEval (Zhang et al., 2024), RepoEval (Zhang et al., 2023), R2E-Eval1, EvoCodeBench (Li et al., 2024a) and DevEval(Li et al., 2024b) consists of function generation tasks in Python and/or Java tasks collected from GitHub repositories. However, they fall short in terms of task complexity and scale. For example, they utilize repositories with limited size, fewer than 243 files on average, significantly smaller than widely-used repositories such as scikit-learn (1,640 files by REPOCOD's collected date).

There is also contemporary work such as SWE-Gym (Pan et al., 2024), which also engages with code at the repository level. However, its primary focus is on training agents for issue fixing, distinguishing it from the code generation emphasis of the benchmarks previously discussed and the objective of REPOCOD.

In contrast, REPOCOD combines complex real-world tasks (analyzed in Section 2.3) from the most popular Python repositories with rigorous test-based assessments and consists of 980 code generation tasks that require LLMs to write large functions instead of single-line snippets, aligning model evaluation with the expectations of modern software development.

Version-Aware Code Generation Benchmarks. Recognizing the dynamic nature of software, another line of benchmarks specifically addresses API version evolution, a factor that can significantly influence code generation performance. LibEvolutionEval (Kuhar et al., 2025) focuses on version-specific in-line code completion and documentation retrieval across library versions. GitChameleon (Islah et al., 2024) provides manually curated Python problems with executable unit tests, conditioned on specific library versions to assess functional accuracy. CodeUpdateArena (Liu et al., 2025) uses synthetic API updates to evaluate knowledge editing in LLMs, testing if models can apply updates without in-context documentation. VersiCode (Wu et al., 2024b) introduces tasks where models must complete code for a specific library version or update code to a different version. It uses a large, diverse dataset and its novel 'Critical Diff Check' metric, designed to specifically assess correct API usage across versions.

While the evolving nature of library versions and its impact on code generation is an important consideration, REPOCOD and the aforementioned version-focused benchmarks evaluate different facets of this challenge. They primarily target the effect of package versions on LLMs' code generation ability, often through simpler tasks such as token/line level completions, specific API updates, or less complex function generation. In contrast, REPOCOD evaluates LLMs' ability to solve complex tasks requiring a deep understanding of repository-level context.

5.3 Automated Test Collection Pipelines

Various approaches have been proposed for collecting benchmarks with repository-level task instances (Zhang et al., 2024, 2023; Jain et al., 2024; Li et al., 2024b). RepoEval executes all tests in the repositories but is impractical for popular repositories due to the sheer volume of test cases. R2E-Eval1, on the other hand, uses LLMs to generate equivalence tests. However, this strategy lacks verification of the correctness of the test cases, limiting its reliability for trustworthy evaluations.

The most comparable approach to ours is proposed in CoderEval, which employs static analysis to identify test cases that reach the target function and supplements them with manually curated test cases. While this approach works for small repositories, it is neither scalable—since it requires human effort—nor does it fully utilize the test cases provided by developers, as static analysis cannot capture all dependencies.

In contrast, our approach uses dynamic analysis to automatically identify test cases from existing developer tests. This execution-based approach is more scalable and is better suited for leveraging existing developer-provided test cases in popular repositories. Additional details on our data collection pipeline are provided in Section 2.1.

6 Conclusion

We present REPOCOD, a real-world, complex dataset designed for code generation tasks with repository-level context. REPOCOD comprises 980 instances from 11 popular Python repositories, including 498 that require repository-level context, with canonical solutions averaging 331 tokens in length—highlighting the benchmark's complexity and comprehensiveness. In addition, We introduce a scalable automatic extraction method for collecting repository-level code generation tasks. Our evaluation of SOTA LLMs on REPOCOD reveals a maximum pass@1 of 28.6%, with even lower scores for functions requiring repository-level context, showing that existing LLMs fall short of generating realistic repository-level code. This work underscores the need for further research in repository-level code generation.

7 Limitation

Our work has limitations. First, we collect data from only 11 repositories, covering a small subset of those that could be included in this benchmark; future versions of REPOCOD will expand to other popular Python repositories. Second, we evaluate only ten models, representing a subset of popular LLMs. With more time and resources, we could test a broader range, providing a more comprehensive view of LLM capabilities in repository-level code generation. We will publish REPOCOD so the community can evaluate additional models, broadening the scope of tested LLMs.

8 Acknowledgements

We thank the anonymous reviewers for their feedback on this work. This research was supported in part by NSF 1901242 and 2006688 and a CFI fund.

References

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

BIG bench authors. 2023. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *Transactions on Machine Learning Research*.

Ning Bian, Xianpei Han, Le Sun, Hongyu Lin, Yaojie Lu, Ben He, Shanshan Jiang, and Bin Dong. 2023. Chatgpt is a knowledgeable but inexperienced solver: An investigation of commonsense problem in large language models. *arXiv preprint arXiv:2303.16421*.

Egor Bogomolov, Aleksandra Eliseeva, Timur Galimzyanov, Evgeniy Glukhov, Anton Shapkin, Maria Tigina, Yaroslav Golubev, Alexander Kovrigin, Arie van Deursen, Maliheh Izadi, and Timofey Bryksin. 2024. Long code arena: a set of benchmarks for long-context code models. *Preprint*, arXiv:2406.11612.

Max Brunsfeld, Andrew Hlynskyi, Amaan Qureshi, Patrick Thomson, Josh Vera, Phil Turnbull, dundargoc, Timothy Clem, ObserverOfTime, Douglas Creager, Andrew Helwer, Rob Rix, Daumantas Kavolis, Hendrik van Antwerpen, Michael Davis, Ika, Tuãn-Anh Nguyễn, Amin Yahyaabadi, Stafford Brunk, Matt Massicotte, Niranjan Hasabnis, bfredl, Mingkai Dong, Samuel Moelius, Steven Kalt, Will Lillis, Kolja, Vladimir Panteleev, and Jonathan Arnett. 2024. tree-sitter/tree-sitter: v0.22.6.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri 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, Josh 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.

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. 2023. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *Preprint*, arXiv:2310.11248.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, et al. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. 2023. Out of the bleu: How should we assess quality of the code generation models? *Journal of Systems and Software*, 203:111741.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming – the rise of code intelligence. *Preprint*, arXiv:2401.14196.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with apps. *NeurIPS*.

Nizar Islah, Justine Gehring, Diganta Misra, Eilif Muller, Irina Rish, Terry Yue Zhuo, and Massimo Caccia. 2024. Gitchameleon: Unmasking the version-switching capabilities of code generation models. *Preprint*, arXiv:2411.05830.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1643–1652, Brussels, Belgium. Association for Computational Linguistics.

Naman Jain, Manish Shetty, Tianjun Zhang, King Han, Koushik Sen, and Ion Stoica. 2024. R2e: Turning any github repository into a programming agent environment. In *ICML*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

Douwe Kiela, Max Bartolo, Yixin Nie, Divyansh Kaushik, Atticus Geiger, Zhengxuan Wu, Bertie Vidgen, Grusha Prasad, Amanpreet Singh, Pratik Ringshia, Zhiyi Ma, Tristan Thrush, Sebastian Riedel, Zeerak Waseem, Pontus Stenetorp, Robin Jia, Mohit Bansal, Christopher Potts, and Adina Williams. 2021. Dynabench: Rethinking benchmarking in NLP. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 4110–4124, Online. Association for Computational Linguistics.

Sachit Kuhar, Wasi Uddin Ahmad, Zijian Wang, Nihal Jain, Haifeng Qian, Baishakhi Ray, Murali Krishna Ramanathan, Xiaofei Ma, and Anoop Deoras. 2025. LibEvolutionEval: A benchmark and study for version-specific code generation. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 6826–6840, Albuquerque, New Mexico. Association for Computational Linguistics.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

Alexander Lex, Nils Gehlenborg, Hendrik Strobelt, Romain Vuillemot, and Hanspeter Pfister. 2014. Upset: Visualization of intersecting sets. *IEEE Transactions on Visualization and Computer Graphics*, 20:1983–1992.

Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2023a. Enabling programming thinking in large language models toward code generation. *arXiv preprint arXiv:2305.06599*.

Jia Li, Ge Li, Xuanming Zhang, Yihong Dong, and Zhi Jin. 2024a. Evocodebench: An evolving code generation benchmark aligned with real-world code repositories. *Preprint*, arXiv:2404.00599.

Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, Lecheng Wang, Kaibo Liu, Zheng Fang, Lanshen Wang, Jiazheng Ding, Xuanming Zhang, Yuqi Zhu, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, Yongbin Li, Bin Gu, and Mengfei Yang. 2024b. DevEval: A manually-annotated code generation benchmark aligned with real-world code repositories. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 3603–3614, Bangkok, Thailand. Association for Computational Linguistics.

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, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour 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. 2023b. Starcoder: may the source be with you! *Preprint*, arXiv:2305.06161.

Yujia Li, David Choi, Junyoung Chung, et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, volume 36, pages 21558–21572. Curran Associates, Inc.

Tianyang Liu, Canwen Xu, and Julian McAuley. 2024. Repobench: Benchmarking repository-level code auto-completion systems. In *The Twelfth International Conference on Learning Representations*.

Zeyu Leo Liu, Shrey Pandit, Xi Ye, Eunsol Choi, and Greg Durrett. 2025. Codeupdatearena: Benchmarking knowledge editing on api updates. *Preprint*, arXiv:2407.06249.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2024. Starcoder 2 and the stack v2: The next generation. *Preprint*, arXiv:2402.19173.

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

T.J. McCabe. 1976. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. *Preprint*, arXiv:2203.13474.

Simon Ott, Adriano Barbosa-Silva, Kathrin Blagec, Janina Brauner, and Matthias Samwald. 2022. Mapping global dynamics of benchmark creation and saturation in artificial intelligence. *Nature Communications*, 13.

Shuyin Ouyang, Jie M Zhang, Mark Harman, and Meng Wang. 2023. Llm is like a box of chocolates: the non-determinism of chatgpt in code generation. *arXiv preprint arXiv:2308.02828*.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. 2024. Training software engineering agents and verifiers with swe-gym. *Preprint*, arXiv:2412.21139.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *Preprint*, arXiv:2009.10297.

Stephen Robertson and Hugo Zaragoza. 2009. The probabilistic relevance framework: Bm25 and beyond. *Found. Trends Inf. Retr.*, 3(4):333–389.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code llama: Open foundation models for code. *Preprint*, arXiv:2308.12950.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti

Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. 2024a. Repoformer: Selective retrieval for repository-level code completion. *Preprint*, arXiv:2403.10059.

Tongtong Wu, Weigang Wu, Xingyu Wang, Kang Xu, Suyu Ma, Bo Jiang, Ping Yang, Zhenchang Xing, Yuan-Fang Li, and Gholamreza Haffari. 2024b. Versicode: Towards version-controllable code generation. *CoRR*, abs/2406.07411.

Chunqiu Steven Xia, Yinlin Deng, and Lingming Zhang. 2024. Top leaderboard ranking= top coding proficiency, always? evoeval: Evolving coding benchmarks via llm. *arXiv preprint arXiv:2403.19114*.

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*, pages 2471–2484, Singapore. Association for Computational Linguistics.

Yakun Zhang, Wenjie Zhang, Dezhi Ran, Qihao Zhu, Chengfeng Dou, Dan Hao, Tao Xie, and Lu Zhang. 2024. Learning-based widget matching for migrating gui test cases. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, volume 66 of *ICSE '24*, page 1–13. ACM.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36:46595–46623.

A Appendix

A.1 Basic Statistics of REPOCOD

REPOCOD contains 980 code generation tasks from 11 widely-used repositories, covering a wide range of functionalities, including data science, scientific computing, web development, and software development tools.

Table 6 presents detailed statistics for instances from each repository, categorized by context complexity types: *repository-level*, *file-level*, and *self-contained*. For each category, it shows **#NL** (number of tokens in target function descriptions), **#GT** (number of tokens in canonical solutions), **Cyclo.** (average cyclomatic complexity of the canonical solution) (McCabe, 1976)), and **#Funcs.** (number of target functions). Additionally, we report repository statistics: **#Line** (average lines in Python files per repository) and **#Files** (average Python files per repository).

We define three types of context complexities: *Repository-level* involves canonical solutions that call functions from other files in the repository; *file-level* involves calls to functions within the same file (excluding the target function) but not from other files; and *self-contained* includes functions that only use commonly used libraries (e.g., `numpy`) or Python built-in modules (e.g., `os`).

Among the three settings, the repository-level functions have the longest token length for the canonical solutions (441.2), compared to file-level functions (192.0) and self-contained functions (233.8). Additionally, the repository-level functions have the highest cyclomatic complexity (10.8) compare to other two categories. The highest length and cyclomatic complexity with the adddditional repository-level context makes repository-level the most challenging category.

A.2 Motivating Example

Figure 6 (a) shows two RepoBench examples where LLM-generated code is semantically correct but wrongly penalized by CodeBLEU, edit similarity, and exact match. In contrast, Figure 6 (b) demonstrates how these metrics assign high scores to incorrect solutions, underscoring their limitations in accurately evaluating generated code.

A.3 Prompting Format

As described in Section 2.2, the data instance of REPOCOD consists of the repository snapshot, target function signature, and docstring. Figure 7 demonstrates an example of the prompt construction process, the format of the prompt, and the evaluation pipeline used for all our experiments.

CodeBLEU: 58.06 Edit Similarity: 85 Exact Match: False

GT:
```
image_paths = [ip for ip in image_paths
               if is_valid_image(ip)]
```

LLM:
```
image_paths = [image_path for image_path in image_paths
               if is_valid_image(image_path)]
```

(a): Semantic equivalent code snippet with low score

CodeBLEU: 76.86 Edit Similarity: 98 Exact Match: False

GT:
```
self.up2 = UpSampleBN(skip_input = 1024 + 64,
                      output_features = 512)
```

LLM:
```
self.up2 = UpSampleBN(skip_input = 1024 + 176,
                      output_features = 512)
```

(b): Incorrect code snippet with high score

Figure 6: Two examples from RepoBench showing misleading metrics results. The yellow and blue highlights indicate the difference between ground truth (GT) and the LLM-generated code.

The section highlighted in green represents the data instance of REPOCOD (step ①). The format of the prompt is detailed in the section highlighted in blue (step ②). Our prompt consists of the system prompt, the file path to the target function (relative to the repository root), the retrieval context, and the target function description (the function signature and docstring of the target function). The retrieval context contains the relative file path, as well as the signature, the docstring, and the body of the retrieved functions. If the context exceeds LLM's context window, we truncate it from the beginning.

Once the LLM generates the solution (step ③), the code snippet is inserted into the repository for testing (step ④, modified file highlighted in blue). Then REPOCOD executes the relevant test cases (step ⑤) and determines the correctness of the solution (step ⑤). As one of the test cases fails, this generated code snippet is considered incorrect.

A.4 Evaluation Result By Repository

Table 7 presents detailed LLM performance on REPOCOD, revealing several key insights. First, repositories differ significantly in difficulty. Models perform well on `flask`, with pass@1 often above 40% , indicating an alignment with LLM capabilities. Second, GPT-4o is the strongest LLM on REPOCOD, consistently outperforming or matching other models across all repositories. Finally, LLMs exhibit specialization across repositories. For instance, while Claude 3.5 Sonnet and DeepSeekCoder-33B perform similarly overall, Claude excels on `pylint`, whereas DeepSeekCoder performs better on `plotly.py`.

Dataset	Repository-level				File-level				Self-contained				Total			
	#NL	#GT	Cyclo.	#Funcs.	#NL	#GT	Cyclo.	#Funcs.	#NL	#GT	Cyclo.	#Funcs.	#NL	#GT	Cyclo.	#Funcs.
astropy	360.6	447.9	10.5	51	404.1	195.7	4.9	14	227.6	238.2	5.9	20	336.5	357.0	8.5	85
datasets	499.7	348.7	9.6	19	292.3	168.8	5.1	18	312.5	128.9	3.1	22	366.6	211.9	5.8	59
flask	277.9	171.9	5.2	8	356.1	160.4	5.2	10	233.0	88.6	3.5	25	270.0	120.8	4.2	43
more-itertools	224.6	70.2	2.6	8	293.4	127.0	4.0	18	243.2	105.2	5.1	60	252.0	106.5	4.6	86
plotly.py	1,357.3	1373.9	37.0	15	1,337.3	356.0	7.9	7	1,687.3	590.5	24.3	54	1,589.9	723.5	25.3	76
pylint	180.9	386.2	14.0	9	163.9	272.3	7.1	7	169.8	178.9	8.0	10	172.0	275.8	9.8	26
scikit-learn	217.9	372.1	7.2	237	299.7	294.9	5.6	27	212.8	237.1	5.5	50	224.1	344.0	6.8	314
seaborn	230.8	538.8	15.0	13	272.7	168.0	4.8	38	163.3	201.1	5.7	27	227.9	241.3	6.8	78
sphinx	201.1	548.4	15.4	14	228.0	240.8	5.5	4	274.1	88.1	3.9	15	237.5	301.9	8.9	33
sympy	874.7	603.4	17.8	67	821.3	182.7	6.5	14	962.4	138.7	4.6	16	881.5	466.0	14.0	97
xarray	791.5	366.9	10.7	57	731.6	102.1	2.8	20	306.2	114.3	2.3	6	742.0	284.8	8.2	83
Average	431.9	441.2	10.8	498	428.1	192.0	5.0	177	528.0	233.8	8.3	305	461.1	331.6	9.0	980

Table 6: Basic statistics of REPOCOD, with details broken down by each collected repository.

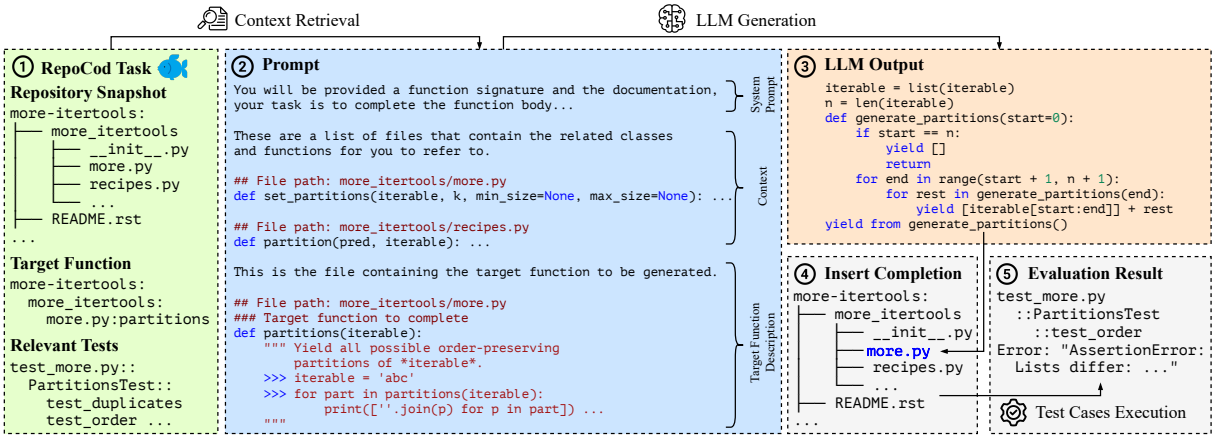


Figure 7: Illustrative example of prompting LLMs and evaluating the LLM-generated code.

A.5 Recall of Retrieval approach

Table 8 presents the recall rates of two different retrieval settings—RAG_{BM25} and $\text{RAG}_{\text{Dense}}$—applied across all selected repositories. These repositories include `pylint`, `sphinx`, `seaborn`, `flask`, `sympy`, `more-itertools`, `scikit-learn`, `xarray`, `datasets`, `plotly.py`, and `astropy`. The recall rates reflect performance at both the repository-level and file-level contexts, providing insights into how each setting performs for individual repositories as well as overall.

The recall rates for $\text{RAG}_{\text{Dense}}$ retrieval are generally higher than those for RAG_{BM25} retrieval across most repositories, highlighting the effectiveness of the $\text{RAG}_{\text{Dense}}$ method in retrieving the invoked functions relevant to the target function.

There are notable variations between repositories. For instance, `more-itertools` and `plotly.py` show significantly higher recall rates with the $\text{RAG}_{\text{Dense}}$ setting, while `scikit-learn` demonstrates relatively low recall rates in both settings.

The total recall rate is computed using micro-averaging, and the result shows that neither retrieval method achieves a high recall for the oracle contexts.

A.6 Inference Length V.S. GT length

We analyze the relationship between inference length and the length of canonical solutions across all retrieval settings. Table 9 reports the average inference-to-canonical solution length ratio using micro-averaging. **On average, LLM-generated results are much longer than canonical solutions.**

We further investigate the causes of excessive generation length and summarize the following contributing factors:

Firstly, it is common for those underperforming LLMs to **generate repetitive code snippets** until they reach the token limit.

Secondly, despite being instructed to generate only the target function, LLMs sometimes **produce additional functions**, resulting in longer outputs. One contributing factor is that when models reference functions not retrieved in the context, they generate the missing helper function definitions, further inflating the output length.

To address this, we apply a filtering method that retains only the first unnested function in the generated output. Specifically, we discard all generated tokens following the first complete function

Setting	pylint	sympy	sphinx	seaborn	flask	more-itertools	scikit-learn	xarray	datasets	plotly.py	astropy	Total
CodeLlama-7B	7.69	7.22	9.09	23.08	37.21	13.95	5.10	6.02	22.03	6.58	5.88	10.41
CodeLlama-34B	0.00	4.12	18.18	21.79	32.56	24.42	5.73	16.87	18.64	14.47	10.59	12.76
DeepSeekCoder-6.7B	7.69	6.19	15.15	26.92	41.86	12.79	6.37	16.87	16.95	25.00	14.12	14.08
DeepSeekCoder-33B	3.85	7.22	27.27	28.21	37.21	11.63	7.32	15.66	33.90	43.42	16.47	17.14
OpenCodeInterpreter-6.7B	3.85	4.12	12.12	16.67	41.86	18.60	6.37	10.84	16.95	21.05	12.94	12.45
OpenCodeInterpreter-33B	11.54	7.22	21.21	20.51	41.86	29.07	8.28	13.25	25.42	23.68	16.47	16.33
Claude 3.5 Sonnet	19.23	11.34	15.15	25.64	39.53	38.37	6.05	9.64	27.12	36.84	10.59	17.45
DeepSeek-V2.5	15.38	13.40	27.27	30.77	41.86	36.05	9.87	16.87	30.51	35.53	16.47	20.71
GPT-4o-Mini	11.54	11.34	15.15	25.64	39.53	18.60	6.05	10.84	33.90	25.00	9.41	15.00
GPT-4o	19.23	15.46	27.27	37.18	58.14	43.02	13.69	24.10	47.46	44.74	23.53	27.04

Table 7: Pass@1(%) of LLMs on REPOCOD under $\text{RAG}_{\text{Dense}}$ retrieval, with details shown for each repository.

Repository	RAG_{BM25}	$\text{RAG}_{\text{Dense}}$
pylint	0.14	0.18
sympy	0.16	0.14
sphinx	0.09	0.11
seaborn	0.17	0.21
flask	0.11	0.20
more-itertools	0.05	0.10
scikit-learn	0.09	0.07
xarray	0.07	0.15
datasets	0.21	0.24
plotly.py	0.02	0.11
astropy	0.17	0.15
Total	0.12	0.15

Table 8: Recall rate of the top three and top ten retrieved contents across retrieval settings for repository-level and file-level contexts.

Models	RAG_{BM25}		$\text{RAG}_{\text{Dense}}$		Current-File	
	Pass	Fail	Pass	Fail	Pass	Fail
CodeLlama-7B	18.2	11.6	9.1	11.9	30.4	22.3
CodeLlama-34B	1.8	15.2	1.9	11.5	22.5	34.5
DeepSeek-6.7B	2.2	9.6	7.0	5.7	20.0	13.1
DeepSeek-33B	8.2	7.4	8.4	3.9	17.9	26.5
OCI-6.7B	2.3	2.5	3.5	2.8	2.5	1.8
OCI-33B	2.4	1.7	2.1	1.5	2.3	1.3
Claude-3.5	19.2	1.7	25.5	1.7	9.8	7.0
DeepSeek-v2.5	7.3	6.4	8.0	7.6	3.6	1.7
GPT-4-Mini	4.0	1.9	4.9	1.8	12.1	1.5
GPT-4	1.7	1.5	1.7	1.5	2.5	1.2

Table 9: Ratio of generation length over canonical solution length

that matches the target function name. Table 10 presents the average filtered inference-to-canonical solution length ratio using micro-averaging. We make two key observations: (1) Failed generations consistently exhibit longer relative lengths after filtering, particularly in Current-File retrieval settings, suggesting that verbosity may correlate with incorrect solutions. (2) LLMs consistently generate longer code snippets than human developers, even when only the first unnested function is considered, indicating a systematic tendency toward verbosity.

Model-Specific Insights.

GPT-4o exhibits similar generation ratios before and after filtering, suggesting it adheres more closely to the task instructions and avoids unnecessary verbosity.

Models	RAG_{BM25}		$\text{RAG}_{\text{Dense}}$		Current-File	
	Pass	Fail	Pass	Fail	Pass	Fail
CodeLlama-7B	0.9	1.7	1.0	1.7	1.0	1.8
CodeLlama-34B	0.9	1.5	0.8	1.4	0.9	1.5
DeepSeek-6.7B	1.0	1.6	0.9	1.4	0.9	1.8
DeepSeek-33B	0.8	1.8	1.0	1.7	0.9	1.9
OCI-6.7B	1.6	2.1	2.8	2.4	1.7	1.5
OCI-33B	1.7	1.2	1.4	1.1	1.4	1.0
Claude-3.5	1.4	1.4	1.6	1.4	1.4	2.1
DeepSeek-v2.5	1.2	1.2	1.1	1.3	1.0	1.4
GPT-4-Mini	1.4	1.1	1.5	1.0	9.1	1.0
GPT-4	1.3	1.3	1.3	1.1	0.9	1.0

Table 10: Ratio of filtered generation length over canonical solution length.

Model	Retrieval	# of Dependencies				
		0	1	2	[3,4]	[5,35]
DeepSeek-v2.5	RAG_{BM25}	31.8	22.1	14.2	7.1	5.7
	$\text{RAG}_{\text{Dense}}$	35.4	28.4	14.2	6.0	6.2
	Current-File	42.6	31.1	22.0	**17.3**	9.1
Claude-3.5	RAG_{BM25}	27.2	18.9	9.9	2.4	2.3
	$\text{RAG}_{\text{Dense}}$	33.1	22.6	12.8	4.8	0.6
	Current-File	25.2	28.4	16.3	14.9	8.5
GPT-4-Mini	RAG_{BM25}	22.6	20.5	14.9	6.0	5.1
	$\text{RAG}_{\text{Dense}}$	24.3	21.1	13.5	5.4	2.8
	Current-File	30.5	23.2	14.9	8.9	5.7
GPT-4	RAG_{BM25}	39.3	32.6	22.7	16.7	**14.8**
	$\text{RAG}_{\text{Dense}}$	**44.6**	**36.3**	16.3	12.5	9.1
	Current-File	39.3	33.7	**24.1**	16.1	10.2

Table 11: Pass Rate by Number of dependencies, binned by the number of unique dependencies. The bins are distributed as follows: 0 (305 instances), 1 (190 instances), 2 (141 instances), [3,4] (168 instances), and [5,35] (176 instances).

Claude-3.5 shows an inverse trend, where successful generations tend to be longer than failed ones (e.g., $\text{RAG}_{\text{Dense}}$: Pass = 25.5 vs. Fail = 1.7). This suggests that Claude-3.5 generates and utilizes helper functions effectively, which may contribute to its success.

A.6.1 Detailed context complexity's impact on Performance

Table 11 demonstrates the relationship between pass@1 and the detailed context complexity (number of unique dependencies) present in the context. Across all models, performance declines as the

Figure 8: Prompt Example for `KMeans.fit` in the file "`sklearn/cluster/_kmeans.py`" from REPOCOD.

number of dependencies increases, highlighting the growing challenge posed by more complex contextual requirements. For instance, DeepSeek-v2.5 with the BM25 retriever achieves a 31.8% pass rate when no dependencies are involved, but this drops to 5.7% when the number of dependencies reaches [5,35].

Notably, retrieval strategies significantly impact performance; models using the Current-File retrieval generally perform better for moderate dependency counts (e.g., DeepSeek-v2.5 achieving 17.3% at [3,4]), suggesting that partial in-file context remains beneficial before complexity outweighs retrieval effectiveness. However, even strong retrieval strategies cannot fully mitigate the difficulty of handling a high number of dependencies, reinforcing the intuition that increasing context complexity directly correlates with reduced task success.

A.7 Case Studies

Prompt Example. Figure 8 shows an example of a REPOCOD prompt, consisting of the system prompt, context information retrieved by RAG_{BM25}, and the target function details, with specific content replaced by '...'.

Passed Example. Figure 9 shows an example of a correctly generated solution from REPOCOD generated by GPT-4o under a $\text{RAG}_{\text{Dense}}$ retrieval setting. This example passes all test cases and is considered correct. Upon inspection, the generated code snippet is identical to the canonical solution except for the text content in the exception message.

A.7.1 Failed Examples

Figure 10 shows an example of an incorrect solution generated by GPT-4o under RAG_{BM25} retrieval setting. The target function `_set_order` is designed to change the order of training data (`X`) and target values (`y`). The canonical solution validates `order` is one of `[None, "C", "F"]`, raising a ValueError if an invalid value is provided. However, for the generated code, it does not validate `order`, which might cause unexpected behavior when `order` is incorrectly specified.

In addition, the canonical solution maintains proper sparse matrix handling by explicitly checking `X` and `y` before conversion, while the generated code can lead to errors if `y` is sparse, since `np.ascontiguousarray()` and `np.asfortranarray()` do not support sparse matrices.

Figure 11 shows another example of an incorrectly generated solution by GPT-4o under RAG_{BM25} retrieval setting. The canonical solution incorporates repository-level context; however, the solution generated by GPT-4o fails to match the functionality of the canonical solution. Specifically, the incorrect solution miscomputes the posterior mean by using an incorrect formulation for the covariance of the latent variables.

A.8 Uniqueness and Overlap of Correct Generations

Commercial LLMs with RAG_{BM25}. Figure 12a presents the UpSet plot comparing the performance of four commercial models using contexts from RAG_{BM25} retrieval, the largest overlap (more than 40) is shared by all models, indicating a significant

Target Function Description

```
def false_alarm_probability(
    self,
    power,
    method="baluev",
    samples_per_peak=5,
    nyquist_factor=5,
    minimum_frequency=None,
    maximum_frequency=None,
    method_kwds=None,
):
    """False alarm probability of periodogram maxima under the null
    hypothesis.

    This gives an estimate of the false alarm probability given the
    height of the largest peak in the periodogram, based on the null
    hypothesis of non-varying data with Gaussian noise.

    Parameters
    ----------
    power : array-like
    The periodogram value.
    method : {'baluev', 'davies', 'naive', 'bootstrap'}, optional
    The approximation method to use.
    maximum_frequency : float
    The maximum frequency of the periodogram.
    method_kwds : dict, optional
    Additional method-specific keywords.

    Returns
    -------
    false_alarm_probability : np.ndarray
    The false alarm probability

    Notes
    -----
    The true probability distribution for the largest peak cannot be
    determined analytically, so each method here provides an
    approximation to the value. The available methods are:

    - "baluev" (default): the upper-limit to the alias-free probability,
    using the approach of Baluev (2008) [1]_.
    - "davies" : the Davies upper bound from Baluev (2008) [1]_.
    - "naive" : the approximate probability based on an estimated
    effective number of independent frequencies.
    - "bootstrap" : the approximate probability based on bootstrap
    resamplings of the input data.

    Note also that for normalization='psd', the distribution can only be
    computed for periodograms constructed with errors specified.

    See Also
    --------
    distribution
    false_alarm_level

    References
    ----------
    .. [1] Baluev, R.V. MNRAS 385, 1279 (2008)
    """
```

Canonical Solution

```
    if self.nterms != 1:
        raise NotImplementedError(
            "false alarm probability is not implemented for "
            "multiterm periodograms."
        )
    if not (self.fit_mean or self.center_data):
        raise NotImplementedError(
            "false alarm probability is implemented "
            "only for periodograms of centered data."
        )

    fmin, fmax = self.autofrequency(
        samples_per_peak=samples_per_peak,
        nyquist_factor=nyquist_factor,
        minimum_frequency=minimum_frequency,
        maximum_frequency=maximum_frequency,
        return_freq_limits=True,
    )
    return _statistics.false_alarm_probability(
        power,
        fmax=fmax,
        t=self._trel,
        y=self.y,
        dy=self.dy,
        normalization=self.normalization,
        method=method,
        method_kwds=method_kwds,
    )
```

LLM Solution

```
    if self.nterms != 1:
        raise NotImplementedError(
            "false alarm level is not implemented for "
            "multiterm periodograms."
        )
    if not (self.fit_mean or self.center_data):
        raise NotImplementedError(
            "false alarm level is implemented "
            "only for periodograms of centered data."
        )

    fmin, fmax = self.autofrequency(
        samples_per_peak=samples_per_peak,
        nyquist_factor=nyquist_factor,
        minimum_frequency=minimum_frequency,
        maximum_frequency=maximum_frequency,
        return_freq_limits=True,
    )
    return _statistics.false_alarm_probability(
        power,
        fmax=fmax,
        t=self._trel,
        y=self.y,
        dy=self.dy,
        normalization=self.normalization,
        method=method,
        method_kwds=method_kwds,
    )
```

Figure 9: An example of the correct output generated by LLMs, for function "`false_alarm_probability`" in the file "`astropy/timeseries/periodograms/lombscargle/core.py`" from REPOCOD.

degree of commonality. GPT-4o has the highest number of unique cases (over 60), suggesting its ability to capture distinct results in this scenario.

Commercial LLMs with Current-File. Figure 12b shows the Current File retrieval setting result. The overlap between all models increases greatly compared to the RAG_{BM25} retrieval (to over 75 cases), demonstrating stronger alignment across models with this retrieval setting. In addition, the unique solvable problem by GPT-4o reduces greatly, to a similar level compared to Claude 3.5 Sonnet and DeepSeek-V2.5.

A.9 Impact of Retrieval Methods on Pass@1 across Varying Context Complexities

Figure 13a and Figure 13b illustrate the pass@1 for instances across various context complexities under different retrieval settings: RAG_{BM25} and current file. In both settings, LLM performance tends to decrease as context complexity rises, progressing from self-contained to file-level and eventually to repository-level contexts. This trend aligns with the insights discussed in Section 4.2.

An interesting finding is that GPT-4-o achieves a higher pass@1 rate for instances under repository-level context complexity compared to file-level instances. This observation suggests two potential insights: (1) specific retrieval methods, like RAG_{BM25}, may enhance performance on repository-level task instances; (2) higher context complexity does not necessarily result in reduced LLM performance.

A.10 REPOCOD-LITE

Given the large number of instances and the high evaluation cost of REPOCOD, we introduce REPOCOD-LITE—a 200 sample subset of the most challenging problems in REPOCOD—enabling faster and more targeted evaluation. We define difficulty based on the product of the prompt

Target Function Description:

```
def _set_order
    """Change the order of X and y if necessary.

    Parameters
    ----------
    X : {array-like, sparse matrix} of shape (n_samples, n_features)
        Training data.

    y : ndarray of shape (n_samples,)
        Target values.

    order : {None, 'C', 'F'}
        If 'C', dense arrays are returned as C-ordered,
        sparse matrices in csr format.
        If 'F', dense arrays are return as F-ordered,
        sparse matrices in csc format.

    Returns
    -------
    X : {array-like, sparse matrix} of shape (n_samples, n_features)
        Training data with guaranteed order.

    y : ndarray of shape (n_samples,)
        Target values with guaranteed order.
    """
```

Instance path:
sklearn/linear_model/_coordinate_descent.py

Canonical Solution

```
if order not in [None, "C", "F"]:
    raise ValueError("Unknown value for order.
    Got {} instead of None, 'C' or 'F'.".format(order))
sparse_X = sparse.issparse(X)
sparse_y = sparse.issparse(y)
if order is not None:
    sparse_format = "csc" if order == "F" else "csr"
if sparse_X:
    X = X.asformat(sparse_format, copy=False)
else:
    X = np.asarray(X, order=order)
if sparse_y:
    y = y.asformat(sparse_format)
else:
    y = np.asarray(y, order=order)
return X, y
```

LLM Solution

```
if sparse.issparse(X):
    if order == "C":
        X = X.tocsr()
    elif order == "F":
    X = X.tocsc()
else:
    X = np.ascontiguousarray(X, dtype=X.dtype) if order == "C"
else np.asfortranarray(X, dtype=X.dtype)
y = np.ascontiguousarray(y, dtype=y.dtype) if order == "C" else
np.asfortranarray(y, dtype=y.dtype)
return X, y
```

Figure 10: A failed case study, for function '_set_order' in the file 'sklearn/linear_model/_coordinate_descent.py' from REPOCOD.

Target Function Description:

```
def transform(self, X):
    """Apply dimensionality reduction to X using the model.

    Compute the expected mean of the latent variables.
    See Barber, 21.2.33 (or Bishop, 12.66).

    Parameters
    ----------
    X : array-like of shape (n_samples, n_features) Training data.

    Returns
    -------
    X_new : ndarray of shape (n_samples, n_components).
    The latent variables of X.
    """
```

Repository-Level Contexsts:
check_is_fitted sklearn/utils/validation.py
validate_data sklearn/utils/validation.py

Canonical Solution

```
    check_is_fitted(self)
    X = validate_data(self, X, reset=False)
    Ih = np.eye(len(self.components_))
    X_transformed = X - self.mean_
    Wpsi = self.components_ / self.noise_variance_
    cov_z = linalg.inv(Ih + np.dot(Wpsi, self.components_.T))
    tmp = np.dot(X_transformed, Wpsi.T)
    X_transformed = np.dot(tmp, cov_z)
    return X_transformed
```

LLM Solution

```
    check_is_fitted(self)
    X = validate_data(self, X, dtype=[np.float64,
                                      np.float32], reset=False)
    # Compute the expected mean of the latent variables
    precision = np.linalg.inv(self.noise_variance_ *
                              np.eye(self.components_.shape[0]) +
                self.components_.T @ self.components_)
    X_new = (X - self.mean_) @ self.components_.T @ precision

    return X_new
```

Figure 11: An example of the wrong output generated by LLMs, for function "transform" in the file "sklearn/decomposition/_factor_analysis.py" from REPOCOD.

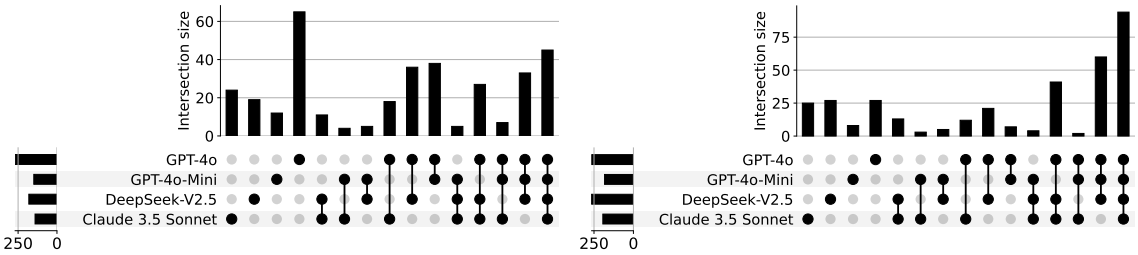
length and canonical solution length (measured in lines of code), selecting samples with the highest scores. REPOCOD-LITE maintains the original benchmark's diversity by including problems from all three categories: self-contained, file-level, and repo-level, with 66, 67, and 67 samples, respectively. This subset enables efficient model evaluation while preserving the complexity of the coding tasks found in the full benchmark. We show the details of REPOCOD-LITE in Table 13. The performance of models on REPOCOD-LITE is shown in Table 12. The SOTA models' performance is consistently lower on REPOCOD-LITE compared to that on REPOCOD.

Models	RAG_{BM25}	RAG_{Dense}	Current-File
CodeLlama-7B	1.0	1.5	1.0
CodeLlama-34B	2.5	2.0	1.0
DeepSeek-6.7B	2.5	2.0	0.5
DeepSeek-33B	4.0	3.5	2.5
OpenCodeInterpreter-6.7B	1.0	5.0	1.5
OpenCodeInterpreter-33B	4.0	6.0	5.5
Claude-3.5	2.5	3.0	2.5
DeepSeek-v2.5	4.0	7.5	5.5
GPT-4o-Mini	8.0	6.5	2.5
GPT-4o	9.0	8.5	4.5

Table 12: Pass@1(%) of SOTA LLMs on REPOCOD-LITE.

A.11 OpenHands's Result on REPOCOD-LITE

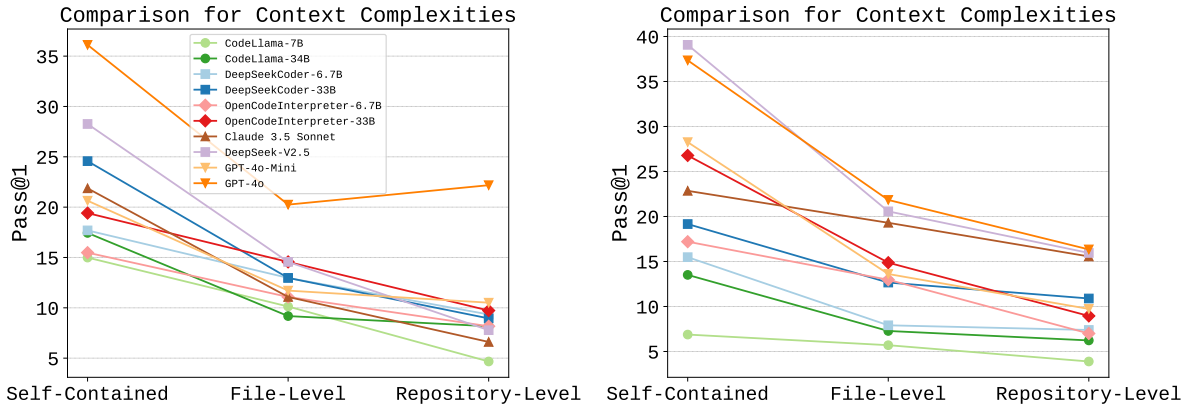
To further explore the performance of advanced code generation approaches on REPOCOD and address the applicability of our benchmark to code



(a) Result for RAG_{BM25}.

(b) Result for Current-File.

Figure 12: Correct generation result's relationship under different retrieval settings.



(a) Pass@1(%) for LLMs based on RAG_{BM25} retrieval.

(b) Pass@1(%) for LLMs based on current file retrieval.

Figure 13: Pass@1 under different context complexity.

agents, we conducted additional experiments using OpenHands, a leading open-source agent framework. For this evaluation, we utilized REPOCOD-LITE

In our experiments, OpenHands, when paired with Claude-3.5 Sonnet as its underlying model, successfully solved 11 out of 200 tasks (5.5% Pass@1) in a single generation round. This result represents an improvement over the performance of Claude-3.5 Sonnet with standard Retrieval-Augmented Generation (RAG) approaches on REPOCOD-LITE.

However, despite this improvement, OpenHands with Claude-3.5 Sonnet still underperformed the best RAG-based result (GPT-4o achieving 9.0% Pass@1 with RAG_{BM25}).

These preliminary findings suggest that while current agentic approaches like OpenHands can offer incremental performance gains on complex, repository-level code generation tasks, they still face significant challenges on REPOCOD. The difficulty of the REPOCOD benchmark, underscores the substantial room for improvement in both LLM capabilities and agentic framework strategies for real-world code generation. We believe these evaluations provide a solid foundation and demonstrate REPOCOD's utility for benchmarking future advancements in code agents.

A.12 Potential Risk and Impact

This work aims to construct an evaluation benchmark for code generation using real-world, repository-level context. We do not foresee any significant risks associated with the misuse of this approach.

Dataset	Repository-level				File-level				Self-contained				Total			
	#NL	#GT	Cyclo.	#Funcs.	#NL	#GT	Cyclo.	#Funcs.	#NL	#GT	Cyclo.	#Funcs.	#NL	#GT	Cyclo.	#Funcs.
astropy	321.7	1,104.0	25.3	3	678.8	1,107.2	20.0	4	485.3	618.8	14.0	9	503.0	831.9	17.6	16
datasets	1,432.8	832.8	24.0	4	1,331.0	526.0	14.0	1	1211.0	318.0	8.0	1	1,378.8	695.8	19.7	6
flask	661.0	424.0	12.0	1	595.0	515.0	12.0	1	452.0	691.0	16.0	1	569.3	543.3	13.3	3
more-itertools	-	-	-	0	503.0	815.0	11.0	1	-	-	-	0	503.0	815.0	11.0	1
plotly.py	1,806.0	3,393.0	132.0	1	1,574.4	1,191.8	30.2	14	1,933.0	1464.2	56.9	23	1,797.5	1,414.6	49.0	38
pylint	-	-	-	0	-	-	-	0	385.0	762.0	16.0	1	385.0	762.0	16.0	1
scikit-learn	420.3	886.3	15.5	51	559.8	727.4	8.0	5	493.6	713.0	15.2	5	437.8	859.1	14.8	61
seaborn	404.3	750.0	18.3	3	-	-	-	0	363.8	678.2	22.5	4	381.1	709.0	20.7	7
sphinx	-	-	-	0	-	-	-	0	418.0	946.0	17.0	1	418.0	946.0	17.0	1
sympy	978.0	424.8	10.8	4	1,135.2	1,155.1	34.5	24	810.2	496.1	14.4	16	1,002.7	849.1	25.0	44
xarray	-	-	-	0	1,562.5	695.1	18.5	17	1,158.8	367.6	13.0	5	1,470.7	620.6	17.2	22
Total	533.2	889.7	17.9	67	1,250.6	987.3	25.7	67	1,120.8	879.0	29.6	66	967.4	918.9	24.39	200

Table 13: Basic statistics of REPOCOD_LITE, with details broken down by each collected repository.