On the Impacts of Contexts on Repository-Level Code Generation

Nam Le Hai , Dung Manh Nguyen, Nghi D. Q. Bui

FPT Software AI Center, Viet Nam

namlh35@fpt.com, dungnm31@fpt.com, bdqnghi@gmail.com

Abstract

CodeLLMs have gained widespread adoption for code generation tasks, yet their capacity to handle repository-level code generation with complex contextual dependencies remains underexplored. Our work underscores the critical importance of leveraging repository-level contexts to generate executable and functionally correct code. We present REPOEXEC, a novel benchmark designed to evaluate repository-level code generation, with a focus on three key aspects: executability, functional correctness through comprehensive test case generation, and accurate utilization of cross-file contexts. Our study examines a controlled scenario where developers specify essential code dependencies (contexts), challenging models to integrate them effectively. Additionally, we introduce an instruction-tuned dataset that enhances CodeLLMs' ability to leverage dependencies, along with a new metric, *Dependency Invocation Rate (DIR)*, to quantify context utilization. Experimental results reveal that while pretrained LLMs demonstrate superior performance in terms of correctness, instruction-tuned models excel in context utilization and debugging capabilities. REPOEXEC offers a comprehensive evaluation framework for assessing code functionality and alignment with developer intent, thereby advancing the development of more reliable CodeLLMs for real-world applications. The dataset and source code are available at https://github.com/FSoft-AI4Code/RepoExec.

1 Introduction

Code Large Language Models (CodeLLMs) have emerged as powerful tools for assisting with coding tasks (Wang et al., 2021, 2023; Feng et al., 2020; Allal et al., 2023; Li et al., 2023; Lozhkov et al., 2024; Guo et al., 2024; Pinnaparaju et al., 2024; Zheng et al., 2024; Roziere et al., 2023; Nijkamp et al., 2022; Luo et al., 2023; Xu et al., 2022; Bui et al., 2023). While these models excel at generating code from natural language requirements or completing individual lines of code, their application in real-world, professional software development scenarios presents more complex challenges. A critical aspect of this complexity lies in leveraging relevant ***contexts*** across an entire software repository, which raises two pivotal questions. First, to what extent are the retrieved dependencies accurate and relevant, rather than potential noise in the input? Second, how effectively do LLMs process and incorporate the provided dependencies into their generated code? These inquiries are central to understanding the capabilities and limitations of CodeLLMs in repository-level code generation, where completing a single line of code might require making API calls to functions within the same file (in-file context) or across different files (cross-file context).

Existing repository-level code generation benchmarks, such as RepoBench (Liu et al., 2023b), RepoCoder (Zhang et al., 2023a), CrossCodeEval (Ding et al., 2023), CoCoMIC (Ding et al., 2024), and DevEval (Li et al., 2024), have made significant strides in assessing code generation at a repository-level scale. However, these benchmarks face several limitations that hinder their ability to comprehensively evaluate real-world coding scenarios:

1. **Lack of an executable environment**: This leads to reliance on match-based metrics, which are not robust for assessing the functional correctness of generated code.

2. **Inadequate control over test quality**: Although RepoCoder and DevEval ensure executability, they depend on pre-existing test cases extracted from repositories, limiting the robustness of evaluation and hindering the scalability of the data pipeline (Liu et al., 2023a).

3. **Overreliance on functional correctness metrics like pass@k**: These metrics, inherited from
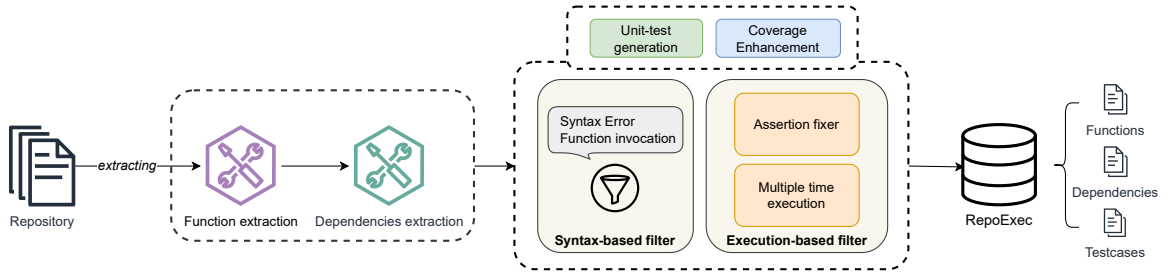
Figure 1: Data Processing Pipeline of REPOEXEC

standalone code generation studies (Chen et al., 2021; Austin et al., 2021), are insufficient in the context of repository-level code generation.

Moreover, real-world development practices often involve breaking down large functions into smaller modules (functions, classes, or variables) to enhance problem-solving efficiency. Consequently, generated code must effectively call these predefined dependencies to align with developer intent. However, LLMs may produce code that passes test cases but relies on inefficient workarounds or unnecessarily reimplements predefined functions. This can lead to suboptimal implementations, code duplication, and potential risks such as technical debt and code smell issues (Maldonado and Shihab, 2015; Sierra et al., 2019; Rasool and Arshad, 2015; Santos et al., 2018).

Given these challenges, it is evident that evaluating CodeLLMs based solely on functional correctness is insufficient. There is a pressing need for a more comprehensive evaluation strategy that assesses not only the correctness of generated code but also its alignment with developer intent, efficiency in utilizing provided dependencies, and adherence to best practices in software development. Crucially, our research reveals a strong correlation between functional correctness (as measured by pass@1) and effective dependency utilization, suggesting that models that better leverage contextual information tend to produce more accurate and functionally correct code.

To effectively address these shortcomings, we introduce a paradigm for evaluation that emphasizes both functional correctness and dependency utilization, while also exploring their relation. Specifically, we introduce the *Dependency Invocation Rate (DIR)*, a novel metric that measures the proportion of code dependencies successfully integrated into the generated code. Higher DIR values indicate better model comprehension of depen-

dencies, while lower values suggest challenges in dependency utilization. *Importantly, our findings demonstrate a strong correlation between DIR and pass@1 rates, underscoring the critical role of effective context utilization in producing high-quality, functionally correct code.*

We also introduce **REPOEXEC**, a novel benchmark specifically designed to overcome the limitations of existing benchmarks, providing a robust and comprehensive evaluation for code generation at real-world applicability. In detail, REPOEXEC advances with two primary focuses:

- **Executability and Dynamic Test Case Generation**: REPOEXEC ensures an executable environment within the repository context, thereby providing a reliable evaluation of functional correctness. Besides, it also incorporates a practical mechanism for dynamically generating high-coverage test cases tailored to the functionality of the newly generated code, ensuring that the code performs its intended tasks accurately.

- **Dependency Usage**: REPOEXEC provides a pipeline to evaluate the usage of dependencies across CodeLLMs, offering insights on how well code dependencies should be provided. Additionally, we introduce an instruction-tuning dataset with dependency calls, which significantly improves CodeLLMs' ability to leverage code dependencies and produce better results.

Our experiments with REPOEXEC reveal key insights into repository-level code generation by CodeLLMs. Models achieve optimal performance with full context dependencies, though smaller contexts outperform medium ones due to formatting issues in the BasePrompt. While pretrained LLMs like Codellama-34b-Python lead in pass@1 rates, instruction-tuned LLMs are better at managing dependencies, despite sometimes generating overly

complex code. In contrast, pretrained LLMs produce accurate code but struggle with dependency utilization. These findings further reinforce the observed correlation between pass@1 and DIR, highlighting the importance of balancing accuracy and context utilization in CodeLLM performance.

Furthermore, we explore two potential approaches for enhancing the quality of generated code: Multi-round debugging and Instruction tuning. Multi-round debugging with test execution, especially with GPT-3.5 and WizardCoder, significantly improves both pass@1 scores and DIR after three rounds. Fine-tuning with our dependency-enhanced dataset not only boosts pass@1 and DIR metrics but also minimizes computational costs, highlighting the importance of executable code testing and the advantages of instruction-tuning and multi-round debugging in enhancing CodeLLMs' performance and dependency management.

In summary, our contributions are as follows:

1. We introduce a novel evaluation paradigm aimed at providing a robust and comprehensive assessment for repository-level code generation. Our approach not only evaluates the functional correctness of generated code but also considers additional quality factors such as maintainability and adherence to clean code principles through effective dependency utilization.

2. We present the Dependency Invocation Rate (DIR), a novel metric that measures the proportion of provided dependencies successfully incorporated into the generated code. This metric helps gauge the models' understanding and utilization of dependencies and shows a strong correlation with functional correctness.

3. We introduce REPOEXEC, a novel benchmark that aligns with our evaluation paradigm, addressing the gaps in existing benchmarks and offering a comprehensive assessment of code generation quality.

4. We implement an effective pipeline within REPOEXEC from dependency extraction, dynamically generate high-coverage test cases and automatic evaluation with execution and code dependencies. Our pipeline offers practical usage and scalability for the community.

5. We release a tool named pydepcall[1] to extract

---
[1] https://github.com/FSoft-AI4Code/pydep/tree/main

dependencies for all functions within any repository, providing a practical use for advancing research in this domain.

6. Our experiments provide crucial insights into the performance of CodeLLMs in repository-level code generation. We demonstrate that while foundation models achieve high initial accuracy, instruction-tuned models excel in dependency management and addressing complex scenarios. Additionally, our multi-round debugging tests show notable improvements in model performance, particularly with enhancements in dependency management. Importantly, we establish a strong correlation between pass@1 and DIR, emphasizing that models which better utilize contexts tend to produce more functionally correct code.

2 Related works

Coding-related tasks have been crucial for assessing the performance of Large Language Models (LLMs), with code generation emerging as a primary focus (Chen et al., 2021; Li et al., 2023; Jiang et al., 2024; Touvron et al., 2023; Roziere et al., 2023; Xu et al., 2022; Allal et al., 2023; Nijkamp et al., 2022). Early benchmarks have been introduced to address this issue (Yin et al., 2018; Iyer et al., 2018; Nguyen et al., 2023; Chen et al., 2021; Austin et al., 2021; Hendrycks et al., 2021); however, they often had limited scope or employed weak evaluation approaches. Some benchmarks (Yin et al., 2018; Iyer et al., 2018; Nguyen et al., 2023) exhibit domain diversity akin to real-world applications; however, their evaluation methodologies are constrained to match-based metrics, thereby decreasing the reliability of these benchmarks (Chen et al., 2021). Meanwhile, benchmarks with reliable evaluation approaches such as HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021) and APPS (Hendrycks et al., 2021) often entail limitations to specific domains like competitive programming. Recently, there have been efforts to extend the domains of generation tasks in various benchmarks. For instance, ExeDS (Huang et al., 2022) focuses on data science code generation, while ODEX (Wang et al., 2022) serves as an open-domain benchmark for code generation. Besides, all the benchmarks mentioned primarily focus on standalone function generation, lacking consideration for cross-contextual and dependency invocation scenarios.

Several recent studies have introduced frameworks and benchmarks for repository-level code generation (Ding et al., 2024; Shrivastava et al., 2023; Ding et al., 2023; Liao et al., 2023; Liu et al., 2023b). These studies closely align with real-world scenarios, underscoring the importance of cross-contextual considerations. However, these works are still limited to match-based evaluation methods. A recent study (Li et al., 2024) introduced the DevEval benchmark for code generation within repository contexts, evaluating performance based on extracted tests available in the repository. These benchmarks primarily focus on assessing the functional correctness of generated outputs but have not extensively investigated the correctness in conjunction with the ability to utilize dependency contexts.

3 Evaluation Paradigm

In this section, we outline our approach to achieve a more robust and comprehensive evaluation of repository-level code generation. Our paradigm encompasses two key attributes: Functional correctness and Dependency utilization.

Functional correctness: This evaluation criterion ensures that the code accurately performs its intended tasks and requirements. Specifically, it typically involves using test cases to compare the execution output of the generated code against the expected output for a given input. This criterion has been widely employed for evaluating code generation in numerous studies (Chen et al., 2021; Austin et al., 2021; Zhuo et al., 2024; Li et al., 2024). We follow the well-known automatic metric *Pass@k* (Chen et al., 2021) to measure the functional correctness of generation outputs.

Dependency utilization: However, functional correctness alone cannot fully assess code quality, as it may overlook issues like poor implementation, workaround solutions, or code redundancy. Assuming that human-written solutions are optimal, match-based metrics like BLEU and edit similarity can assess the alignment between the generated code and the high-quality reference solution at the token, word, or character level. However, not all words and tokens contribute to the quality of the code and alternative implementations can preserve the quality despite low similarity scores. For instance, local variable names or comments can vary widely without impacting code quality. Additionally, statements like `while` and `for` can be interchangeable while preserving the

algorithm's complexity. Thus, these match-based metrics are also not reliable for accurately measuring code quality. Meanwhile, tokens representing called dependencies—such as third-party packages, functions, variables, and classes within the repository—demonstrate effective use of quality code for efficient implementation. In contrast, ignoring these dependencies may suggest workaround implementations misaligned with human intent, leading to extensive verification and maintenance costs.

Therefore, to assess the models' ability to utilize provided dependencies in accordance with human intent, we introduce the *Dependency Invocation Rate (DIR)*. This metric represents the percentage of invoked dependencies out of the total number of dependencies provided. Let D_g denote the set of identifiers in the generated output, and D_s denote the set of provided dependencies extracted from the solution. The Dependency Invocation Rate (DIR) is calculated as follows:

$$\text{DIR} = \frac{|D_g \cap D_s|}{|D_s|}$$

A higher DIR indicates that the model successfully incorporates a larger proportion of the provided dependencies into the generated code, demonstrating a better understanding of the dependencies' relevance and their intended usage. Conversely, a lower DIR suggests that the model may struggle to identify and utilize the appropriate dependencies, potentially generating code that is less aligned with the human-written solution.

In summary, we state that achieving a high-quality solution requires the generated code to excel in both functional correctness and dependency utilization. Otherwise, it indicates a lack of ability to generate correct solutions for the intended tasks or suggests poor implementation practices, including technical debt or code smell issues.

4 Data Collection Pipeline

4.1 Data Source

Developing an executable benchmark within repository contexts is challenging due to complex setup requirements and frequent lack of clear installation guidelines in repositories. Previous studies (Ding et al., 2024; Shrivastava et al., 2023; Ding et al., 2023; Liao et al., 2023) have often relied on match-based metrics for evaluation, which may not fully capture code functionality. In addition, test cases

errors.py

```
...

(1) class InvalidInputError(TypeError):

(2)     """
        Custom error raised when received object is not
        a string as expected.
        """

(1)     def __init__(self, input_data: Any):

(2)         """
            :param input_data: Any received object
            """

(3)         type_name = type(input_data).__name__
            msg = 'Expected "str", received "
                    {}"'.format(type_name)
            super().__init__(msg)
...
```

Components legend:

| (1) dependency signature | (2) dependency docstring | (3) dependency content | (4) dependency variable | (5) in-file import | (6) Target function prompt |

Prompt

Full context

(5) + (4) + ((1) + (2) + (3)) + (6)

Medium context

(5) + (4) + ((1) + (2)) + (6)

Small context

(5) + (4) + (1) + (6)

Cross-file context

validation.py

```
...

(4) DEFAULT_SEPERATOR = '_'

(1) def is_string(obj: Any) -> bool:

(2)     """
        Checks if an object is a string.

        *Example:*

        >>> is_string('foo') # returns true
        >>> is_string(b'foo') # returns false

        :param obj: Object to test.
        :return: True if string, false otherwise.
        """

(3)     return isinstance(obj, str)


(1) def is_camel_case(input_string: Any) -> bool:

(2)     """
        Checks if a string is formatted as camel case.

        A string is considered camel case when:

        - it's composed only by letters ([a-zA-Z]) and
        optionally numbers ([0-9])
        - it contains both lowercase and uppercase letters
        - it does not start with a number

        *Examples:*

        >>> is_camel_case('MyString') # returns true
        >>> is_camel_case('mystring') # returns false

        :param input_string: String to test.
        :type input_string: str
        :return: True for a camel case string, false otherwise.
        """

(3)     return is_full_string(input_string) and
            CAMEL_CASE_TEST_RE.match(input_string) is
            not None
...
```

In-file context

manipulation.py

```
(5) import base64
    import random
    import unicodedata
    import zlib
    from typing import Union
    from uuid import uuid4
    from ._regex import *
    from .errors import InvalidInputError
    from .validation import is_snake_case,
    is_full_string, is_camel_case, is_integer, is_string,
    DEFAULT_SEPERATOR

(4) CAMEL_CASE_REPLACE_RE = re.compile(r'([a-z]|[A-Z]+)(?
    =[A-Z])')

(6) def camel_case_to_snake(input_string,
    separator=DEFAULT_SEPERATOR):
        """
        Convert a camel case string into a snake case one.
        (The original string is returned if is not a valid
        camel case string)

        *Example:*

        >>> camel_case_to_snake('ThisIsACamelStringTest') #
        returns 'this_is_a_camel_case_string_test'

        :param input_string: String to convert.
        :type input_string: str
        :param separator: Sign to use as separator.
        :type separator: str
        :return: Converted string.
        """

        if not is_string(input_string):
            raise InvalidInputError(input_string)

        if not is_camel_case(input_string):
            return input_string

        return CAMEL_CASE_REPLACE_RE.sub(lambda m:
            m.group(1) + separator,
            input_string).lower()
...
```
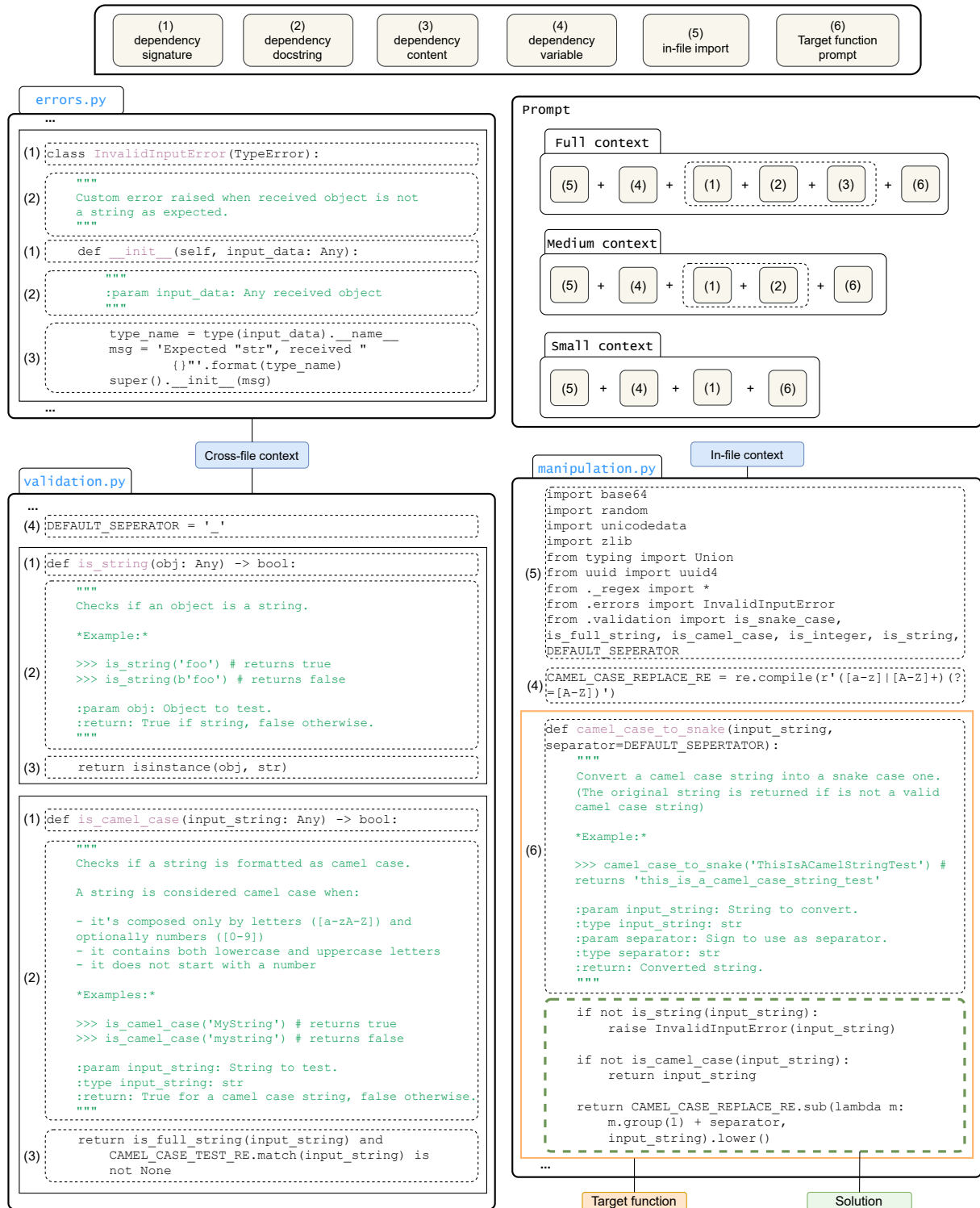
Target function   Solution

Figure 2: Illustration of a data instance in RepoExec. The target function signatures and their associated docstrings, which describe the functionality of the functions, are shown in (6). The infile-imports and variable declarations are represented by (5) and (4), respectively. The remaining components, (1), (2), and (3), represent the function and class contexts. Specifically, (1) denotes the class or function signature, (2) may contain the description of the class, and (3) represents the function body of the cross-file function.

are essential for assessing code functionality. How-   ever, extracting test cases from repositories (Zhang

et al., 2023b; Li et al., 2024; Zhang et al., 2024) often relies on available functions and heuristic rules, limiting adaptability and excluding data without existing tests. For instance, Li et al. 2024 found that over 99% of functions were discarded due to the absence of suitable tests. We propose a dataset collection pipeline to ensure that repositories can build executable environments and that test cases are automatically generated.

4.2 Functions and Dependencies Extraction

Function extraction: We extract functions and their dependencies from repositories, considering only those suitable for function-level code generation, akin to benchmarks like HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021). Using tree-sitter, we parse files into Abstract Syntax Trees (AST) to extract functions, focusing on those with comprehensive docstrings and excluding functions used as entry points or that do not produce verifiable outputs.

Dependencies extraction: We employ static analysis to identify dependencies, excluding identifiers that are function parameters or typing objects. Each dependency name is then mapped to its definition within the repository using a repository graph and static analysis tools. We release our tool pydepcall for community usage and provide a brief description in Appendix H.

For example in Figure 2, for the function camel_case_to_snake in manipulation.py, our process identifies CAMEL_CASE_REPLACE_RE as an in-file dependency and analyzes import statements to track cross-file dependencies from errors.py and validation.py. Dependencies are then parsed and incorporated into the input prompt to ensure comprehensive context for code generation.

4.3 Test case generation

To overcome the limitation of requiring available tests in the repository for evaluation (as mentioned in Section 4.1), we leverage large language models (LLMs) to generate test cases automatically. Our proposed approach ensures the correctness of created test cases while also controlling and enhancing test coverage. To execute and validate the generated test cases, it is necessary to configure each repository to create an executable environment. Follow Lemieux et al. 2023, we use pipreqs[2] to identify each project's dependencies.

In our test generation process, we conduct two phases corresponding to **Correctness Control** and **Coverage Enhancement**. After the test generation process, we exclude samples with a line coverage lower than 40%, as they are of insufficient quality to accurately assess the correctness of the generated code.

4.3.1 Correctness Control

In this stage, we present the procedure for generating initial test cases and ensure the tests' correctness. Specifically, we use CodeLlama-13b (Roziere et al., 2023) and provide the model with the prompt detailed in Appendix C. After generation, we extract the first 20 assertion statements to construct the test cases. To ensure their correctness, we employ syntax and execution-based filtering methods to exclude low-quality test cases.

Syntax-based filter: We filter out tests that present syntax errors during parsing into AST. Additionally, tests that do not invoke the function under test are excluded. For instance, while the statement "assert 1" may pass during execution, it is meaningless and negatively impacts the evaluation.

Execution-based filter: We use pytest[3] to execute the generated tests. Tests are discarded if their execution output displays an error, except for AssertionError. If an assertion error is observed, we try to fix it using the Assertion Fixer.

- **Assertion Fixer:** The fixer resolves this error by executing the test case with the target function, extracting the output, and reassigning it to the assertion statement. Given that the return type can be complex in the repository (e.g., a defined object), we use pickle[4] to preserve the execution results.

- **Multiple time execution:** We witness some flaky tests, which produce inconsistent outcomes upon multiple executions due to the inherent randomness in their implementation. To eliminate these instances, we execute each test 10 times to compare the outputs.

4.3.2 Coverage Enhancement

Weak unit tests may allow incorrect implementations to pass (Liu et al., 2023a). To address this, we propose a strategy for enhancing test coverage

---

[2]https://github.com/bndr/pipreqs

[3]https://github.com/pytest-dev/pytest
[4]https://docs.python.org/3/library/pickle.html

using LLMs. Given the complexity of this task, requiring a strong understanding of code, we utilize GPT-3.5 to improve the quality of test cases. We provide GPT-3.5 with three prompts (see Appendix C) to handle challenging scenarios, including edge and corner cases. The initial generated tests serve as few-shot examples. We then extract the newly generated tests and ensure their correctness using our methodology from Section 4.3.1. Table 1 shows a line coverage improvement of about 4% after enhancement, reaching 96.25%. The performance gap has also increased to over 5% (Appendix B), indicating greater robustness in the evaluation.

5 Data Characteristics

5.1 Data Formatting

Figure 2 illustrates the input data format used in REPOEXEC. We retain import information and append dependencies in the order presented in the import statements. The target function signature and natural language description are positioned at the end of the input prompt. We propose three prompt types with varying context lengths to test the reasoning capability of CodeLLMs in leveraging contexts for repo-level code generation:

- **Full-size context**: All contexts, including cross-file and in-file contexts, are preserved to assess the model's ability to navigate and utilize the complete information available in the repository.

- **Medium-size context**: Class and function bodies are removed, while their signatures and docstrings are retained. This tests the model's ability to infer the functionality and usage of dependencies based on their interfaces and documentation, reducing the input context length.

- **Small-size context**: Only the signatures of the dependencies are retained. This tests if CodeLLMs can infer the usage of dependencies in the target function given only the function signatures without docstrings, representing the most challenging case with minimal information.

Evaluating the model's performance across these context sizes provides insights into the trade-offs between input context length and the model's reasoning capabilities, helping to understand the optimal balance between providing sufficient information and minimizing input size for effective code generation at the repository level.

Dataset	#Samples	Context	TC	LC (%)
CoNaLA (Yin et al., 2018)	500	✗	✗	-
CONCODE (Iyer et al., 2018)	2,000	✗	✗	-
HumanEval (Chen et al., 2021)	164	✗	✓, **H**	99.43
MBPP (Austin et al., 2021)	974	✗	✓, **H**	98.48
RepoCoder (Zhang et al., 2023b)	373	✓	✓, **P**	-
CrossCodeEval (Ding et al., 2023)	2,665	✓	✗	-
CoCoMIC (Ding et al., 2024)	6,888	✓	✗	-
DevEval (Li et al., 2024)	1,874	✓	✓, **P**	-
REPOEXEC	355	✓	✓, **A**	92.46
+ coverage-enhancement				96.25

Table 1: The comparison between popular code generation benchmarks and REPOEXEC. For test case, we denote **H**: Human annotated, **P**: Pre-existing, **A**: Automated. **TC**: Test Cases. **LC**: Line Coverage

We follow Muennighoff et al. 2023 to define 2 types of prompt formats in our evaluation of REPOEXEC across LLMs: (1) BasePrompt, which concatenates all contexts with the target functions (Figure 2), and (2) InstructPrompt, which includes specific instructions for the LLMs to follow, utilizing two variations as input formats (further details and examples in Appendix D).

5.2 Dataset Stastistic

Comparison to Existing Benchmarks:

Table 1 compares the details of REPOEXEC with those of existing code generation benchmarks. Benchmarks that exclude execution-based evaluation (Yin et al., 2018; Iyer et al., 2018; Ding et al., 2023, 2024) may gather substantial amounts of data; however, they are inadequate for assessing the quality of the generated code. For HumanEval and MBPP, the majority of problems involve standalone functions, which do not reflect real-world scenarios. Besides, benchmarks that rely on human-annotated and pre-existing test cases (Chen et al., 2021; Austin et al., 2021; Zhang et al., 2023a; Li et al., 2024) are challenging to scale and control the test coverage. Finally, repository-context benchmarks (RepoCoder (Zhang et al., 2023a), CrossCodeEval (Ding et al., 2023), CoCoMIC (Ding et al., 2024), and DevEval (Li et al., 2024)) primarily focus on evaluating retrieval modules. For example, RepoCoder introduces a retriever using a sparse bag-of-words model and provides the effectiveness of this retriever. Similarly, CrossCodeEval experiments and reports performance using various retrievers such as BM25, UniXCoder, and OpenAI ada. CoCoMic proposed CCFINDER to retrieve cross-file context. DevEval is the closest to our work; however, they compare the performance of the generation model based on different given types

	#No problem		#No testcase		Avg tokens	
	Cross-file	Total	Avg LC (%)	Avg	Prompt	Solution
Full	22.8%	355	96.25	99.45	362.92	78.46
Medium	-	-	-	-	253.05	-
Small	-	-	-	-	179.66	-

Table 2: Dataset attributes with different levels of contexts (Full, Medium, Small). Cross-file refers to the percentage of problems that involve cross-file dependencies. Tree-sitter is used for tokenization. **AVG LC**: Average Line Coverage.

of context, which can also align with the different contexts provided by different retrievers. In contrast, our approach emphasizes the generation module's ability to understand and utilize human-provided dependency contexts.

Dataset attributes: Table 2 outlines the characteristics of REPOEXEC, including the total number of examples, the average number of test cases per problem, and the number of tokens in prompts and solutions.

6 Experiment

6.1 Evaluation Results

We evaluated 13 CodeLLMs on REPOEXEC and presented the results (pass@1, pass@5, and DIR) in Table 3. We use nucleus sampling with temperature set to 0.2, top-p to 0.95, and 10 outputs generated for all models. The results show that retaining the full context of dependencies yields the best performance across all models. Surprisingly, Small-size context proves to be more effective than Medium-size context, which we attribute to the context's input format using BasePrompt, potentially misleading the model into interpreting the dependency functions as few-shot examples. Using the Small-size context results in a fair decrease in performance compared to the Full context while effectively reducing the input length by half.

Codellama-34b-Python achieves the highest pass@1 rate among the models considered, and foundation models demonstrate greater effectiveness compared to instruction-tuned models on our dataset. However, our analysis reveals inherent limitations in both types of models:

1. Instruction-tuned LLMs demonstrate a higher capacity for utilizing given dependencies than foundation LMs, sometimes enabling them to address edge or corner cases that foundation models have overlooked.

2. Despite the high DIR, instruction-tuning LMs

may not utilize dependencies correctly and frequently produce overly complex code, leading to incorrect solutions.

3. Pretrained LLMs produce code that functions correctly but do not effectively reuse the provided dependencies, occasionally duplicating the implementation from the given context. This leads to redundancy and potentially creates technical debt or code smell issues (Maldonado and Shihab, 2015; Sierra et al., 2019; Rasool and Arshad, 2015; Santos et al., 2018).

These issues can lead to a low-quality codebase, requiring substantial human effort for reviewing and fixing, which may even exceed the effort needed to write the code from scratch. Details and examples are discussed further in Appendix E.

6.2 Generation with Multi-round Debugging

In this section, we examine the models' self-refinement capabilities in enhancing generation performance. We provide the models with error output logs and ask them to fix the errors. We experiment with WizardCoder, GPT3.5, and CodeLlama-13b-Python. The number of rounds to debug is set to 3 and the input template is presented in Appendix F. In this experiment, we employ a greedy search algorithm to generate only a single output.

Table 4 shows the improvement across three rounds of debugging in various models. We observe that GPT-3.5 and WizardCoder demonstrate a high capacity for debugging with improvement of over 10% and 7% in pass@1, respectively, while CodeLlama fails to take advantage of this process. Additionally, the DIR has also shown a significant improvement (over 7%) after three rounds of debugging in these two instruction models (Figure 7). These findings indicate a promising approach using self-refinement with debugging for code generation, which can enhance both the correctness and the utilization of given dependencies.

6.3 Instruction-tuning with Code Dependencies

While the multi-round debugging experiment has demonstrated effectiveness in leveraging given dependencies to provide correct solutions (Section 6.2), it requires a strong model to generate good test cases and can be time-consuming due to the repeated generation and execution of code and test cases. To address these challenges, we propose

	Model	Full context			Medium context			Small context		
		pass@1	pass@5	DIR	pass@1	pass@5	DIR	pass@1	pass@5	DIR
	BasePrompt									
Pre-models	CodeLlama-34b-Python (Roziere et al., 2023)	**42.93**	**49.54**	**68.85**	**35.92**	**42.95**	58.15	**39.80**	**45.79**	64.23
	CodeLlama-13b-Python (Roziere et al., 2023)	<u>38.65</u>	<u>43.24</u>	62.26	<u>32.96</u>	<u>38.33</u>	56.38	<u>35.66</u>	<u>42.41</u>	62.67
	StarCoder (Li et al., 2023)	28.08	33.95	58.75	22.54	31.83	50.74	25.54	31.45	56.67
	StarCoder2-15b (Lozhkov et al., 2024)	27.77	32.60	60.57	18.70	23.28	39.28	23.27	29.78	53.49
	Mixtral-8x7B-v0.1 (Jiang et al., 2024)	22.82	29.14	55.90	19.38	25.25	47.22	20.54	26.30	53.40
	Phi-2 (Javaheripi et al., 2023)	19.04	24.56	48.22	14.54	20.34	40.85	14.82	20.69	44.54
	Phi-1 (Gunasekar et al., 2023)	14.99	18.38	43.17	12.48	15.42	37.45	12.54	15.96	38.75
Inst-models	WizardCoder-Python-13B-V1.0 (Luo et al., 2023)	34.31	40.06	62.90	30.99	36.75	59.50	32.54	38.34	64.67
	Phind-CodeLlama-34B-v2	30.08	33.49	59.47	25.25	29.40	50.73	27.55	31.85	58.93
	CodeLlama-13b-Instruct (Roziere et al., 2023)	28.56	32.67	57.09	26.25	30.72	49.48	26.73	33.50	54.53
	GPT-3.5	27.27	37.69	63.59	23.15	33.94	52.79	22.59	33.63	55.22
	DeepSeek-Coder-7b-Instruct (Guo et al., 2024)	25.18	29.91	58.50	20.23	26.02	45.76	22.20	27.74	56.69
	Mixtral-8x7B-Instruct-v0.1 (Jiang et al., 2024)	23.41	28.71	59.83	19.04	24.83	52.75	20.45	25.84	58.17
	InstructPrompt									
Inst-models	WizardCoder-Python-13B-V1.0	26.20	30.68	67.32	24.56	30.25	<u>67.54</u>	24.70	29.56	<u>68.34</u>
	CodeLlama-13b-Instruct	25.66	30.82	62.04	27.44	34.11	63.33	26.73	32.43	64.85
	GPT-3.5	23.82	39.10	40.55	19.62	36.03	37.48	19.00	34.00	35.28
	Mixtral-8x7B-Instruct-v0.1	18.11	23.04	<u>67.73</u>	18.54	23.12	**69.66**	15.38	20.61	**68.86**

Table 3: Pass@k (k= 1 and 5) and DIR results of various LLMs on REPOEXEC. **Bold scores** indicate the highest values, while <u>Underlined scores</u> represent the second highest. Pre- and Inst- denote Pretrained and Instruction-tuned, respectively.

Round	GPT-3.5	WizardCoder	CodeLlama-13b-Python
0	27.04	34.37	39.44
1	36.34	40.85	39.44
2	40.00	41.69	39.44
3	41.97	42.54	39.44

Table 4: Pass@1 scores of various models across three rounds of debugging. Round 0 represents the initial generation stage.

Model	Full context		Small context	
	Pass@1	DIR	Pass@1	DIR
phi-2	19.04	48.22	14.82	44.54
phi-2$_{DepIT}$	**20.20**	**61.66**	**20.31**	**70.30**
StarCoder	28.08	58.67	25.49	56.67
StarCoder$_{DepIT}$	**29.43**	**69.80**	**28.73**	**71.48**
StarCoder2	27.77	60.57	23.27	53.49
StarCoder2$_{DepIT}$	**28.45**	**69.76**	**27.27**	**73.98**
CodeLlama	38.65	62.26	35.66	62.67
CodeLlama$_{DepIT}$	**38.85**	**68.89**	**36.93**	**73.19**

Table 5: Comparison of the performance of several models on REPOEXEC after instruction tuning for dependency calls ($_{DepIT}$) with their pre-trained versions.

an instruction-tuning dataset for fine-tuning base LLMs. We collected the 1,555 most-starred repositories from 2018 onward and extracted functions with their corresponding dependencies, following the procedure outlined in Section 4.2. We obtained 154,818 functions, of which 57,746 samples include docstrings. From this set, we utilized 50K samples with docstrings and applied instruction prompts, while 80K samples adhered to the raw code format (Full context). Recognizing the potential of the Small context format, we allocated the remaining 20K samples to follow this structure. We fine-tuned Phi-2, StarCoder, StarCoder2, and CodeLlama-13b-Python models for 5 epochs with LoRa (Hu et al., 2021) and used 10% of the training data as the validation set to select the best checkpoint.

Table 5 illustrates the efficacy of our training data. All 4 models demonstrate improvements in both Pass@1 and DIR after instruction tuning. Specifically, there is a slight increase of around 1% in Pass@k for all models, while DIR shows a significant improvement, reaching the highest

scores (over 70%) compared to other models after tuning with our dataset. Notably, the performance improves significantly with the use of small context, achieving results comparable to those obtained with full context. This offers the potential for more efficient processing and reducing computational costs, allowing additional space to integrate various other types of context. In summary, our proposed instruction-tuning method can improve the model's ability to both reuse dependencies and ensure the functional correctness of the output. Although self-refinement through multi-round debugging (Section 6.2) demonstrates greater effectiveness, instruction-tuning models only utilize single-turn generation, making them more efficient in practice. *To facilitate open research in this domain, we will publicly release this dataset.*

7 Conclusion

We introduce an evaluation approach for repository-level code generation that rethinks the limitations of previous methods by assessing not only functional correctness but also dependency utilization to ensure the quality of the generated code. By presenting REPOEXEC, a novel Python code generation benchmark at the repository level with executable capabilities, designed to evaluate the alignment of generated code with developer intent and its correctness. Our experiments reveal that while pretrained language models (LLMs) excel in functional correctness, instruction-tuned models demonstrate proficiency in utilizing dependencies and debugging. However, existing models struggle to effectively reuse provided dependencies, potentially leading to technical debt and code smell issues. We provide a comprehensive evaluation of how different LLMs leverage code dependencies when generating code, and our findings show that the size of the context significantly impacts the final results.

We also introduce an instruction-tuning dataset that enhances dependency invocation accuracy and output correctness, even with limited context. This approach allows for the integration of additional context types and mitigates large token length constraints. Our contributions establish a foundation for future research in code generation, providing valuable evaluation techniques to drive the development of more capable and reliable models.

8 Limitations

REPOEXEC is constructed for evaluating models based on existing works. However, this approach can potentially lead to data leakage, especially when modern models are trained on similar datasets. If the benchmark relies heavily on known works, there's a risk that the model may inadvertently learn from specific patterns or features present in those works, compromising its generalization ability. Despite this concern, experimental results demonstrate that even modern models struggle to handle the challenges posed by REPOEXEC, indicating that the benchmark remains a valuable tool for assessing model performance. Besides, several models, including StarCoder2 and DeepSeek-Coder, have been pretrained using repository-level context. However, these models typically concatenate the contents of multiple files in a repository without filtering out irrelevant information or considering the selection of dependencies. This helps our dataset distinguish from the pretraining datasets of these models, thereby helping to mitigate the data leakage issue. However, employing new repositories can help mitigate these phenomena.

In this work, we only consider one level of dependency context, specifically the dependencies directly called from the target function. While this simplification facilitates manageable analysis and model development, it may not fully capture the valuable context necessary for leveraging models effectively. However, incorporating deep dependencies could significantly extend the input length, posing challenges in managing long context inputs and potentially exceeding the maximum input length. Our approach has demonstrated promising outcomes with the small-size context version, creating opportunities for integrating additional input context. Future research could explore incorporating multiple levels of dependencies, creating a more comprehensive graph that includes transitive dependencies, indirect calls, and broader contextual information. By doing so, we could enhance the model's understanding of code interactions and improve its ability to handle intricate software execution scenarios.

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*.

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*.

Nghi DQ Bui, Hung Le, Yue Wang, Junnan Li, Akhilesh Deepak Gotmare, and Steven CH Hoi. 2023. Codetf: One-stop transformer library for state-of-the-art code llm. *arXiv preprint arXiv:2306.00029*.

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

Yangruibo Ding, Zijian Wang, Wasi U. Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2024. CoCoMIC: Code completion by jointly modeling in-file and cross-file context. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*.

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. 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*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. 2023. Textbooks are all you need. *arXiv preprint arXiv:2306.11644*.

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

Rajarshi Haldar and Julia Hockenmaier. 2024. Analyzing the performance of large language models on code summarization. *arXiv preprint arXiv:2404.08018*.

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*.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.

Junjie Huang, Chenglong Wang, Jipeng Zhang, Cong Yan, Haotian Cui, Jeevana Priya Inala, Colin Clement, Nan Duan, and Jianfeng Gao. 2022. Execution-based evaluation for data science code generation models. *arXiv preprint arXiv:2211.09374*.

Maor Ivgi, Uri Shaham, and Jonathan Berant. 2023. Efficient long-text understanding with short-text models. *Transactions of the Association for Computational Linguistics*, 11:284–299.

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.

Mojan Javaheripi, Sébastien Bubeck, Marah Abdin, Jyoti Aneja, Sebastien Bubeck, Caio César Teodoro Mendes, Weizhu Chen, Allie Del Giorno, Ronen Eldan, Sivakanth Gopi, et al. 2023. Phi-2: The surprising power of small language models. *Microsoft Research Blog*.

Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. 2024. Mixtral of experts. *arXiv preprint arXiv:2401.04088*.

Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 919–931. IEEE.

Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, Lecheng Wang, Kaibo Liu, Zheng Fang, Lanshen Wang, et al. 2024. Deveval: A manually-annotated code generation benchmark aligned with real-world code repositories. *arXiv preprint arXiv:2405.19856*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Dianshu Liao, Shidong Pan, Qing Huang, Xiaoxue Ren, Zhenchang Xing, Huan Jin, and Qinying Li. 2023. Context-aware code generation framework for code repositories: Local, global, and third-party library awareness. *arXiv preprint arXiv:2312.05772*.

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. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173.

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*.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

Everton da S Maldonado and Emad Shihab. 2015. Detecting and quantifying different types of self-admitted technical debt. In *2015 IEEE 7Th international workshop on managing technical debt (MTD)*, pages 9–15. IEEE.

Debanjan Mondal, Abhilasha Lodha, Ankita Sahoo, and Beena Kumari. 2023. Robust code summarization. In *Proceedings of the 1st GenBench Workshop on (Benchmarking) Generalisation in NLP*, pages 65–75, Singapore. Association for Computational Linguistics.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. 2023. Octopack: Instruction tuning code large language models. *arXiv preprint arXiv:2308.07124*.

Dung Nguyen, Le Nam, Anh Dau, Anh Nguyen, Khanh Nghiem, Jin Guo, and Nghi Bui. 2023. The vault: A comprehensive multilingual dataset for advancing code understanding and generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 4763–4788, Singapore. Association for Computational Linguistics.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

Nikhil Pinnaparaju, Reshinth Adithyan, Duy Phung, Jonathan Tow, James Baicoianu, Ashish Datta, Maksym Zhuravinskyi, Dakota Mahan, Marco Bellagente, Carlos Riquelme, et al. 2024. Stable code technical report. *arXiv preprint arXiv:2404.01226*.

Ghulam Rasool and Zeeshan Arshad. 2015. A review of code smell mining techniques. *Journal of Software: Evolution and Process*, 27(11):867–895.

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

José Amancio M Santos, João B Rocha-Junior, Luciana Carla Lins Prates, Rogeres Santos Do Nascimento, Mydiã Falcão Freitas, and Manoel Gomes De Mendonça. 2018. A systematic review on the code smell effect. *Journal of Systems and Software*, 144:450–477.

Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023. Repository-level prompt generation for large language models of code. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 31693–31715. PMLR.

Giancarlo Sierra, Emad Shihab, and Yasutaka Kamei. 2019. A survey of self-admitted technical debt. *Journal of Systems and Software*, 152:70–82.

Ankita Nandkishor Sontakke, Manasi Patwardhan, Lovekesh Vig, Raveendra Kumar Medicherla, Ravindra Naik, and Gautam Shroff. 2022. Code summarization: Do transformers really understand code? In *Deep Learning for Code Workshop*.

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*.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*.

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*.

Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2022. Execution-based evaluation for open-domain code generation. *arXiv preprint arXiv:2212.10481*.

Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 1–10.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of the 15th international conference on mining software repositories*, pages 476–486.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023a. Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570*.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023b. 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.

Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. *arXiv preprint arXiv:2401.07339*.

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*.

Appendix

A   Evaluation Metrics: Match-Based vs. Execution-Based

Several code generation benchmarks utilize match-based metrics like Edit similarity (ES), BLEU, and CodeBLEU for evaluation (Ding et al., 2024; Shrivastava et al., 2023; Ding et al., 2023; Liao et al., 2023; Yin et al., 2018; Iyer et al., 2018). These metrics are straightforward to apply and may exhibit a strong correlation with execution metrics such as Pass@k. However, they cannot accurately measure functional correctness. For instance, comparing two Python code snippets where the only difference is the ":" character could result in a good ES and BLEU score. Nevertheless, one snippet may contain a syntax error, highlighting a limitation in these metrics for assessing true functionality.

Figure 3: Correlation between Match-based metrics and Execution-based metric (Pass@1).

Figure 3 demonstrates that these metrics on average can achieve a strong correlation with Pass@k, with CodeBLEU showing the highest correlation with Pearson score of 0.92. However, upon closer inspection of the score distribution between correct and incorrect solutions (Figure 4), a considerable overlap becomes apparent. This underscores the limitation of match-based metrics in accurately measuring the correctness of code generation.

Figure 4: Match-based metric distributions between Correct and Incorrect solutions

B   Coverage Enhancement Effectiveness

Weak unit tests may inadvertently allow incorrect implementations to be determined as correct. Even with human-written tests, the overlooked coverage rates lead to evaluations that are incomplete and potentially misleading (Liu et al., 2023a). We present evidence supporting this argument, underscoring the limitations of prior work on code generation within repository-level contexts. As depicted in Figure 5, enhancing the

number of test cases and coverage rates leads to a significant increase in the identification of incorrect generated solutions, causing the Pass@1 score to drop markedly (by over 5%). We investigated several solutions and found that most of the generated results did not fully utilize the given context (considered as human-provided). Instead, they primarily focused on addressing the problem described in the given natural language description. This indirectly overlooks the developer's intentions, such as testing edge or corner cases, highlighting the limitations in following and understanding the provided intent and dependency context in these models. In summary, these findings underscore the effectiveness and importance of maintaining high-quality test cases for evaluation purposes.

Figure 5: Performance of various CodeLMs on REPOEXEC before (bf-) and after (af-) Coverage Enhancement (CovEn) stage.

C Test case generation

```
Initial Test Generation Prompt

{function_under_test}
# test to check the correctness of "{function_name}" function
assert
```

Coverage Enhancement Prompts

Prompt 1:
Here are some Python unit test functions and the focal function that they test:
Test functions:
{existing_test_functions}
Focal function:
{function_under_test}
Write more unit test functions that will increase the test coverage of the function under test.

Prompt 2:
Here are some Python unit test functions and the focal function that they test:
Test functions:
{existing_test_functions}
Focal function:
{function_under_test}
Write more unit test functions that will cover corner cases missed by the original and will increase the test coverage of the function under test.

Prompt 3:
Here is a focal function under test:
{function_under_test}
This function under test can be tested with these Python unit test functions:
{existing_test_functions}
Here is an extended version of the unit test function that includes additional unit test cases that will cover methods, edge cases, corner cases, and other features of the function under test that were missed by the original unit test functions:

D   Data Formating

D.1   BasePrompt

<div style="border:1px solid #ccc">

Example of Full Context

```python
import base64
import random
import unicodedata
import zlib
from typing import Union
from uuid import uuid4
from ._regex import *
from .errors import InvalidInputError
from .validation import is_snake_case, is_full_string, is_camel_case, is_integer, is_string

CAMEL_CASE_REPLACE_RE = re.compile(r'([a-z]|[A-Z]+)(?=[A-Z])')

class InvalidInputError(TypeError):
    """
    Custom error raised when received object is not a string as expected.
    """

    def __init__(self, input_data: Any):
        """
        :param input_data: Any received object
        """
        type_name = type(input_data).__name__
        msg = 'Expected "str", received "{}"'.format(type_name)
        super().__init__(msg)

def is_string(obj: Any) -> bool:
    """
    Checks if an object is a string.

    *Example:*

    >>> is_string('foo') # returns true
    >>> is_string(b'foo') # returns false

    :param obj: Object to test.
    :return: True if string, false otherwise.
    """
    return isinstance(obj, str)

def is_camel_case(input_string: Any) -> bool:
    """
    Checks if a string is formatted as camel case.

    A string is considered camel case when:

    - it's composed only by letters ([a-zA-Z]) and optionally numbers ([0-9])
    - it contains both lowercase and uppercase letters
    - it does not start with a number

    *Examples:*

    >>> is_camel_case('MyString') # returns true
    >>> is_camel_case('mystring') # returns false

    :param input_string: String to test.
    :type input_string: str
    :return: True for a camel case string, false otherwise.
    """
    return is_full_string(input_string) and CAMEL_CASE_TEST_RE.match(input_string) is not None

def camel_case_to_snake(input_string, separator='_'):
    """
    Convert a camel case string into a snake case one.
    (The original string is returned if is not a valid camel case string)

    *Example:*

    >>> camel_case_to_snake('ThisIsACamelStringTest') # returns 'this_is_a_camel_case_string_test'

    :param input_string: String to convert.
    :type input_string: str
    :param separator: Sign to use as separator.
    :type separator: str
    :return: Converted string.
    """
```

</div>

Example of Medium Context

```python
import base64
import random
import unicodedata
import zlib
from typing import Union
from uuid import uuid4
from ._regex import *
from .errors import InvalidInputError
from .validation import is_snake_case, is_full_string, is_camel_case, is_integer, is_string

CAMEL_CASE_REPLACE_RE = re.compile(r'([a-z]|[A-Z]+)(?=[A-Z])')

class InvalidInputError(TypeError):
    """
    Custom error raised when received object is not a string as expected.
    """

    def __init__(self, input_data: Any):
        """
        :param input_data: Any received object
        """

def is_string(obj: Any) -> bool:
    """
    Checks if an object is a string.

    *Example:*

    >>> is_string('foo') # returns true
    >>> is_string(b'foo') # returns false

    :param obj: Object to test.
    :return: True if string, false otherwise.
    """

def is_camel_case(input_string: Any) -> bool:
    """
    Checks if a string is formatted as camel case.

    A string is considered camel case when:

    - it's composed only by letters ([a-zA-Z]) and optionally numbers ([0-9])
    - it contains both lowercase and uppercase letters
    - it does not start with a number

    *Examples:*

    >>> is_camel_case('MyString') # returns true
    >>> is_camel_case('mystring') # returns false

    :param input_string: String to test.
    :type input_string: str
    :return: True for a camel case string, false otherwise.
    """

def camel_case_to_snake(input_string, separator='_'):
    """
    Convert a camel case string into a snake case one.
    (The original string is returned if is not a valid camel case string)

    *Example:*

    >>> camel_case_to_snake('ThisIsACamelStringTest') # returns 'this_is_a_camel_case_string_test'

    :param input_string: String to convert.
    :type input_string: str
    :param separator: Sign to use as separator.
    :type separator: str
    :return: Converted string.
    """
```

Example of Small Context

```python
import base64
import random
import unicodedata
import zlib
from typing import Union
from uuid import uuid4
from ._regex import *
from .errors import InvalidInputError
from .validation import is_snake_case, is_full_string, is_camel_case, is_integer, is_string

CAMEL_CASE_REPLACE_RE = re.compile(r'([a-z]|[A-Z]+)(?=[A-Z])')

class InvalidInputError(TypeError):

    def __init__(self, input_data: Any):

def is_string(obj: Any) -> bool:

def is_camel_case(input_string: Any) -> bool:

def camel_case_to_snake(input_string, separator='_'):
    """
    Convert a camel case string into a snake case one.
    (The original string is returned if is not a valid camel case string)

    *Example:*

    >>> camel_case_to_snake('ThisIsACamelStringTest') # returns 'this_is_a_camel_case_string_test'

    :param input_string: String to convert.
    :type input_string: str
    :param separator: Sign to use as separator.
    :type separator: str
    :return: Converted string.
    """
```

D.2 InstructPrompt

Instruction Prompt Templates

Prompt 1:

```
### Instruction:

Write a Python function `{target_function_signature}` to solve the following
    problem:
{target_function_docstring}

### Response:
{BasePrompt}
```

_____

Prompt 2:

```
### Instruction:

{dependency_context}
The provided code snippet includes necessary dependencies for implementing the
    `{target_function_name}` function. Write a Python function `{
    target_function_signature}` to solve the following problem:
{target_function_docstring}

### Response:
{target_function_prompt}
```

```
### Instruction:

Write a Python function `camel_case_to_snake(input_string, separator='_')` to solve the following
    problem:
"""
Convert a camel case string into a snake case one.
(The original string is returned if is not a valid camel case string)

*Example:*

>>> camel_case_to_snake('ThisIsACamelStringTest') # returns 'this_is_a_camel_case_string_test'

:param input_string: String to convert.
:type input_string: str
:param separator: Sign to use as separator.
:type separator: str
:return: Converted string.
"""

### Response:
import base64
import random
import unicodedata
import zlib
from typing import Union
from uuid import uuid4
from ._regex import *
from .errors import InvalidInputError
from .validation import is_snake_case, is_full_string, is_camel_case, is_integer, is_string

CAMEL_CASE_REPLACE_RE = re.compile(r'([a-z]|[A-Z]+)(?=[A-Z])')

class InvalidInputError(TypeError):

    def __init__(self, input_data: Any):

def is_string(obj: Any) -> bool:

def is_camel_case(input_string: Any) -> bool:

def camel_case_to_snake(input_string, separator='_'):
    """
    Convert a camel case string into a snake case one.
    (The original string is returned if is not a valid camel case string)

    *Example:*

    >>> camel_case_to_snake('ThisIsACamelStringTest') # returns 'this_is_a_camel_case_string_test'

    :param input_string: String to convert.
    :type input_string: str
    :param separator: Sign to use as separator.
    :type separator: str
    :return: Converted string.
    """
```

Example of Prompt 2 for Small Context

```
### Instruction

import base64
import random
import unicodedata
import zlib
from typing import Union
from uuid import uuid4
from ._regex import *
from .errors import InvalidInputError
from .validation import is_snake_case, is_full_string, is_camel_case, is_integer, is_string

CAMEL_CASE_REPLACE_RE = re.compile(r'([a-z]|[A-Z]+)(?=[A-Z])')

class InvalidInputError(TypeError):

    def __init__(self, input_data: Any):

def is_string(obj: Any) -> bool:

def is_camel_case(input_string: Any) -> bool:

The provided code snippet includes necessary dependencies for implementing the `camel_case_to_snake`
      function. Write a Python function `camel_case_to_snake(input_string, separator='_')` to solve
      the following problem:
"""
Convert a camel case string into a snake case one.
(The original string is returned if is not a valid camel case string)

*Example:*

>>> camel_case_to_snake('ThisIsACamelStringTest') # returns 'this_is_a_camel_case_string_test'

:param input_string: String to convert.
:type input_string: str
:param separator: Sign to use as separator.
:type separator: str
:return: Converted string.
"""

### Response:
def camel_case_to_snake(input_string, separator='_'):
    """
    Convert a camel case string into a snake case one.
    (The original string is returned if is not a valid camel case string)

    *Example:*

    >>> camel_case_to_snake('ThisIsACamelStringTest') # returns 'this_is_a_camel_case_string_test'

    :param input_string: String to convert.
    :type input_string: str
    :param separator: Sign to use as separator.
    :type separator: str
    :return: Converted string.
    """
```

E   Studied LLMs: Supplemental results

In this section, we offer supplementary results from the evaluation of LLMs on REPOEXEC. Table 3 presents the Dependency Invocation Rate (DIR) for the experimented LLMs. When comparing models of the same size, it is shown that instruction-tuned models more effectively follow human intent in utilizing the provided dependencies with InstructPrompt. For example, WizardCoder outperforms CodeLlama by 5%, and the instruction-tuned version of Mixtral-8x7B shows a 10% improvement over its foundation version. This highlights the strong capability of instruction-tuned models to follow the given context effectively. Besides, using the Medium context leads to a significant decline in both Pass@k and DIR using BasePrompt. This implies the generation of empty function bodies using this template.

Indeed, Figure 6 illustrates the proportion of generated functions that are empty for each LLM using BasePrompt. The findings indicate that utilizing Medium context results in a substantial number of empty functions. This may be due to the input format of the context when using BasePrompt, which can mislead the model into interpreting dependency functions as few-shot examples. In the Medium context, the function bodies of dependencies are removed, making their format identical to the target function
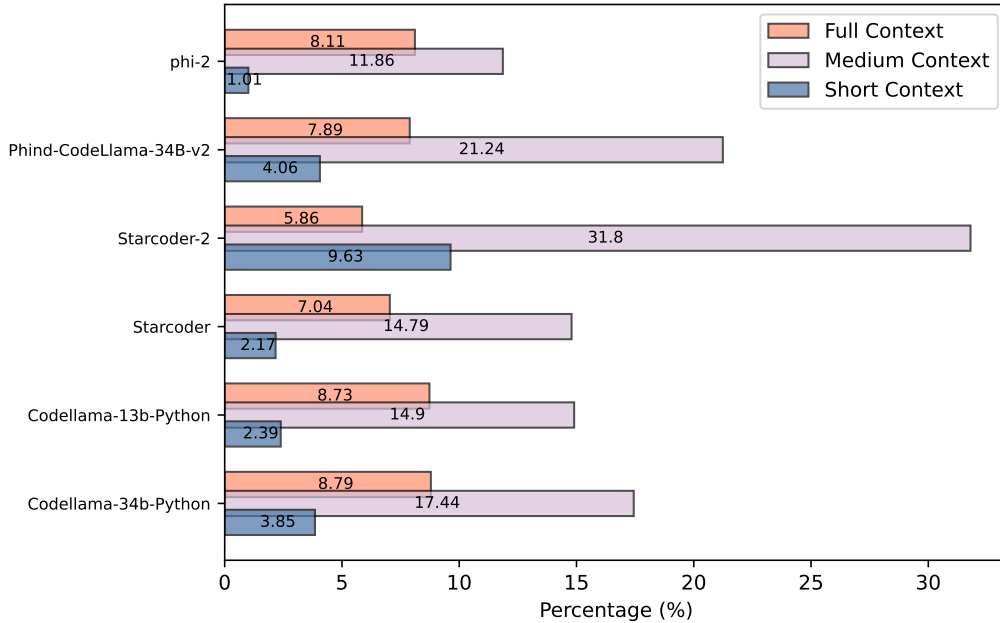
Figure 6: Percentage of generated outputs that result in empty functions across various context types.

prompt. This similarity can mislead the LMs, resulting in empty solutions. Particularly, Starcoder-2 is heavily impacted by this issue, as over 31% of its generated results are empty functions, revealing a significant weakness of the model. Meanwhile, small context effectively decreases the occurrence of empty function generation by the model and, in certain instances, improves models' ability in dependency calls (e.g. CodeLlama-13b-Python, WizardCoder-Python-13B-V1.0, and Mixtral-8x7B-Instruct-v0.1 in Table 3). We believe that the following reasons could contribute to this observation. Firstly, employing small context reduces the input token count, preventing truncation when exceeding the maximum length limit, thus allowing uninterrupted solution generation by the model. This reduced context enables models to concentrate exclusively on dependency signatures, thereby enhancing the probability that generated solutions effectively utilize these dependency token names. Moreover, function names hold substantial semantic value by delineating the function's purpose. Many studies have underscored that code summarization heavily relies on extracting information from function names (Haldar and Hockenmaier, 2024; Mondal et al., 2023; Sontakke et al., 2022). Therefore, this concise representation of dependencies has the potential to improve how models utilize dependencies in generating code.

Additionally, Table 6 presents examples that support our findings in Section 6.1. For instance, in the first example, we observe that the instruction-tuned model can effectively utilize the given dependencies to manage edge cases, whereas the pretrained model fails to do so. This supports our initial findings. Meanwhile, the second and third examples demonstrate that pretrained models often reimplement or devise workarounds instead of leveraging the available context. Besides, in the second example, the instruction-tuned model correctly identifies the relevant case but fails to generate the correct solution. This may suggest that hallucinations complicate the outputs generated by instruction-tuned models.

F   Multi-round Debugging

We employ Multi-round debugging in code generation, which iteratively refines and improves the generated code through multiple cycles of debugging. Following the execution of unit tests on the generated functions, we extract the error log if the code fails to run. We employ the following prompt to utilize the model for bug fixing. This process is iterated multiple times until either the correct code is achieved or the maximum number of rounds is reached. Specifically, we set the maximum number of rounds to 3 and experimented on three models WizardCoder, GPT-3.5 and CodeLlama-13b-Python.
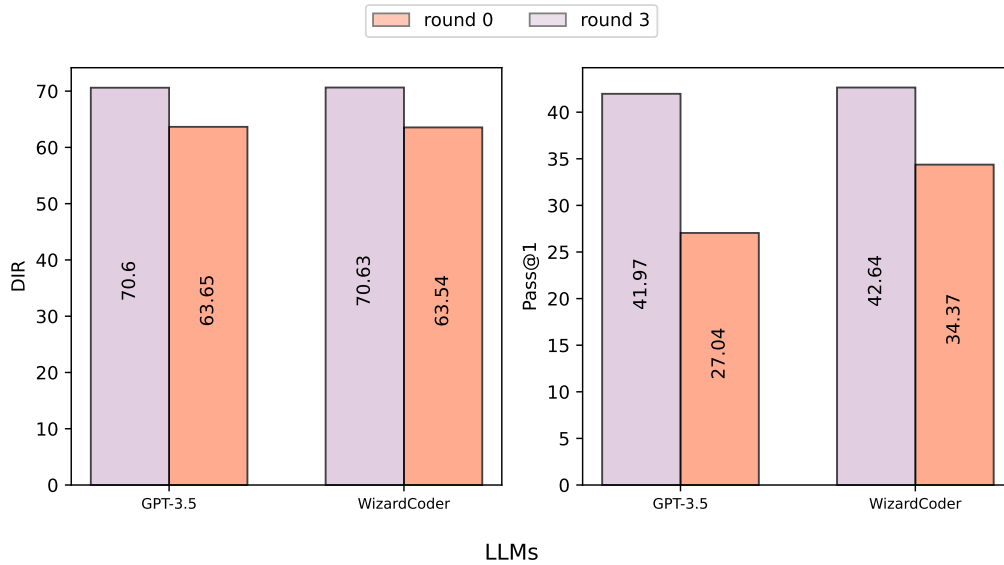
Figure 7: Improvement of instruction-tuning models on Pass@1 and DIR after 3-round debugging process.

```
Prompt for Debugging

{dependency_context}
# The provided code snippet includes necessary dependencies for implementing
    the `{target_function_name}` function. Write a Python function `{
    target_function_signature}` to solve the following problem:
{target_function_docstring}

# Here is the current solution.
{error_solution}

# When executing the below test case.
{failed_test_case}

# The provided python code solution fails the test with the following errors,
    please correct them.
{error_log}

# Please provide the modified code for me to review and provide feedback.
{target_function_prompt}
```

Table 4 shows the improvement across three rounds of debugging in various models. We observe that GPT-3.5 and WizardCoder demonstrate a high capacity for debugging with improvement of over 10% and 7% in Pass@1, respectively, while CodeLlama fails to take advantage of this process. Additionally, the DIR has also shown a significant improvement (over 7%) after three rounds of debugging in these two instruction models (Figure 7). These findings indicate a promising approach using self-refinement with debugging for code generation, which can enhance both the correctness and the utilization of given dependencies.

We also present data on the number of error types corrected in each round of WizardCoder, as illustrated in Figure 8. We can see that `AssertionError` makes up the majority of errors across all rounds. This error type indicates either incorrect outputs from the generated code or the presence of empty function bodies (return None). However, by incorporating the test output guide, the model effectively addressed most of these errors. Furthermore, fundamental issues like `SyntaxError` or `AttributeError` were promptly rectified during the initial round.
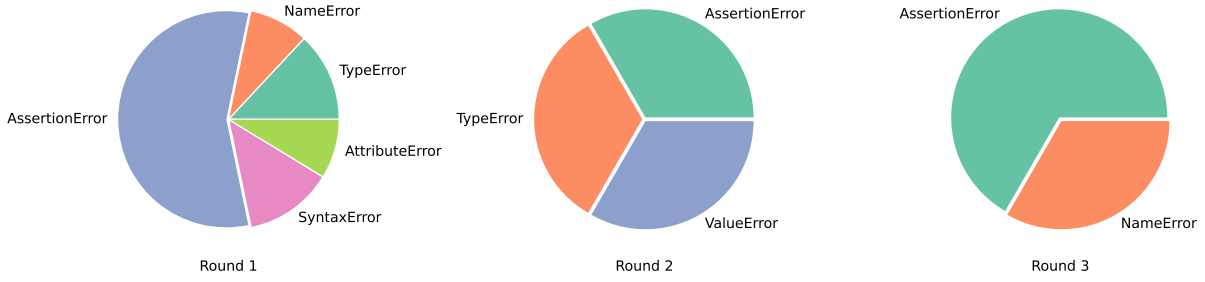
Figure 8: Fixed error types of WizardCoder across 3 rounds of the debugging process.

G Context length analysis

Long-context models can enhance the ability to comprehend and select relevant context from lengthy inputs to effectively solve the required task. In this section, we examine how the context length supported by each model affects their performance in REPOEXEC. Figure 9 demonstrates the relationship between support context length, model size, and model family in relation to pass@k and DIR scores. The size of the dots indicates the model size, while the color represents the model family.
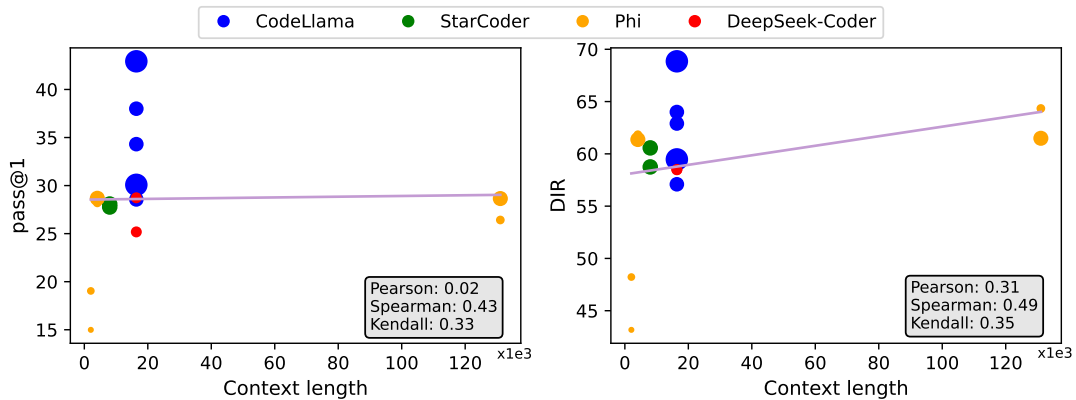

Figure 9: Correlation of context length to the model performance on REPOEXEC.

We observe that context length has a weak correlation with model performance on our dataset. In contrast, model size (scaling law) and model family, which encompass different training methods (pretraining or instruction tuning), training datasets, and architectures, show a more significant impact. The weak correlation with context length can be explained by our approach's ability to already capture relevant information (e.g. dependency) for each data sample, resulting in the pruning of context length for practical usage (363 tokens on average shown in Table 2). Meanwhile, models that support long contexts are often trained on data containing a mix of relevant and irrelevant information and are evaluated on their ability to retrieve the correct context in a needle-in-a-haystack scenario (Roziere et al., 2023; Ivgi et al., 2023; Liu et al., 2024). Therefore, models with varying context lengths might show a weak correlation to performance in our scenario.

H Dependency extraction tool usage

We present pydepcall, a Python library designed to extract function dependencies from any repository. We provide a brief overview of its usage in the following code snippet.

pydepcall usage

```python
from pydepcall import Extractor

# If you want to extract all module files in the repository
reposrc = YOUR_LOCAL_PATH_OF_REPO
extractor = Extractor(reposrc)
output = extractor.extract()

# If you want to extract a specific module file in the repository
reposrc = YOUR_LOCAL_PATH_OF_REPO
module_file = YOUR_LOCAL_PATH_OF_FILE_IN_REPO
extractor = Extractor(reposrc, module_file)
output = extractor.extract()
```

Input Prompt & Solution	**Generation output**

```
# Prompt:
import base64
import random
import unicodedata
...

class InvalidInputError(TypeError):
    ...

def is_string(obj: Any) -> bool:
    ...

def reverse(input_string: str) -> str:
    """
    Returns the string with its chars reversed.

    *Example:*

    >>> reverse('hello') # returns 'olleh'

    :param input_string: String to revert.
    :type input_string: str
    :return: Reversed string.
    """

--------------------------------------------------------

# Solution:
def reverse(input_string: str) -> str:
    """
    Returns the string with its chars reversed.

    *Example:*

    >>> reverse('hello') # returns 'olleh'

    :param input_string: String to revert.
    :type input_string: str
    :return: Reversed string.
    """
    if not is_string(input_string):
        raise InvalidInputError(input_string)

    return input_string[::-1]
```

```
# Model type: Pretrained
# Passed: No
def reverse(input_string: str) -> str:
    """
    Returns the string with its chars reversed.

    *Example:*

    >>> reverse('hello') # returns 'olleh'

    :param input_string: String to revert.
    :type input_string: str
    :return: Reversed string.
    """
    return input_string[::-1]

--------------------------------------------------------

# Model type: Instruction-tuned
# Passed: Yes
def reverse(input_string: str) -> str:
    """
    Returns the string with its chars reversed.

    *Example:*

    >>> reverse('hello') # returns 'olleh'

    :param input_string: String to revert.
    :type input_string: str
    :return: Reversed string.
    """
    if not is_string(input_string):
        raise InvalidInputError(input_string)
    return input_string[::-1]
```

```
# Prompt:
import random
import luigi
import luigi.configuration
...

class hadoopcli(luigi.Config):
    command = ...
    version = ...

def get_configured_hadoop_version():
    """
    CDH4 (hadoop 2+) has a slightly different syntax
     for interacting with hdfs
    via the command line.

    The default version is CDH4, but one can override
    this setting with "cdh3" or "apache1" in the
     hadoop section of the config
    in order to use the old syntax.
    """
--------------------------------------------------------

# Solution:
def get_configured_hadoop_version():
    """
    CDH4 (hadoop 2+) has a slightly different syntax
     for interacting with hdfs
    via the command line.

    The default version is CDH4, but one can override
    this setting with "cdh3" or "apache1" in the
     hadoop section of the config
    in order to use the old syntax.
    """
    return hadoopcli().version.lower()
```

```
# Model type: Pretrained
# Passed: Yes
def get_configured_hdfs_client():
    """
    This is a helper that fetches the configuration
     value for 'client' in
    the [hdfs] section. It will return the client that
     retains backwards
    compatibility when 'client' isn't configured.
    """
    return luigi.configuration.get_config().get('hdfs'
    , 'client', 'hadoopcli')

--------------------------------------------------------

# Model type: Instruction-tuned
# Passed: No
def get_configured_hadoop_version():
    """
    CDH4 (hadoop 2+) has a slightly different syntax
     for interacting with hdfs
    via the command line.

    The default version is CDH4, but one can override
    this setting with "cdh3" or "apache1" in the
     hadoop section of the config
    in order to use the old syntax.
    """
    hadoop_cli = hadoopcli().command.split()
    hadoop_version = hadoopcli().version
    if hadoop_version == "cdh3":
        return hadoop_cli + ["fs"]
    elif hadoop_version == "apache1":
        return hadoop_cli + ["dfs"]
    else:
        return hadoop_cli
```

```
# Prompt:
import asyncio
import builtins
import collections
...

def is_future(x: Any) -> bool:
    return isinstance(x, FUTURES)

def _create_future() -> Future:
    future = Future()  # type: Future
    ...

def maybe_future(x: Any) -> Future:
    """Converts ``x`` into a `.Future`.

    If ``x`` is already a `.Future`, it is simply
     returned; otherwise
    it is wrapped in a new `.Future`.  This is
     suitable for use as
    ``result = yield gen.maybe_future(f())`` when you
     don't know whether
    ``f()`` returns a `.Future` or not.

    .. deprecated:: 4.3
    This function only handles ``Futures``, not other
     yieldable objects.
    Instead of `maybe_future`, check for the non-
     future result types
    you expect (often just ``None``), and ``yield``
     anything unknown.
    """
----------------------------------------------------

# Solution:
def maybe_future(x: Any) -> Future:
    """Converts ``x`` into a `.Future`.

    If ``x`` is already a `.Future`, it is simply
     returned; otherwise
    it is wrapped in a new `.Future`.  This is
     suitable for use as
    ``result = yield gen.maybe_future(f())`` when you
     don't know whether
    ``f()`` returns a `.Future` or not.

    .. deprecated:: 4.3
      This function only handles ``Futures``, not
    other yieldable objects.
      Instead of `maybe_future`, check for the non-
    future result types
       you expect (often just ``None``), and ``yield``
     anything unknown.
    """
    if is_future(x):
         return x
    else:
        fut = _create_future()
        fut.set_result(x)
        return fut
```

```
# Model type: Pretrained
# Passed: Yes
def maybe_future(x: Any) -> Future:
    """Converts ``x`` into a `.Future`.

    If ``x`` is already a `.Future`, it is simply
     returned; otherwise
    it is wrapped in a new `.Future`.  This is
     suitable for use as
    ``result = yield gen.maybe_future(f())`` when you
     don't know whether
    ``f()`` returns a `.Future` or not.

    .. deprecated:: 4.3
    This function only handles ``Futures``, not other
     yieldable objects.
    Instead of `maybe_future`, check for the non-
     future result types
    you expect (often just ``None``), and ``yield``
     anything unknown.
    """
    if is_future(x):
        return x
    future = Future()  # type: Future
    future.set_result(x)
    return future

----------------------------------------------------

# Model type: Instruction-tuned
# Passed: Yes
def maybe_future(x: Any) -> Future:
    """Converts ``x`` into a `.Future`.

    If ``x`` is already a `.Future`, it is simply
     returned; otherwise
    it is wrapped in a new `.Future`.  This is
     suitable for use as
    ``result = yield gen.maybe_future(f())`` when you
     don't know whether
    ``f()`` returns a `.Future` or not.

    .. deprecated:: 4.3
    This function only handles ``Futures``, not other
     yieldable objects.
    Instead of `maybe_future`, check for the non-
     future result types
    you expect (often just ``None``), and ``yield``
     anything unknown.
    """
    if isinstance(x, Future):
        return x
    elif isawaitable(x):
        return asyncio.ensure_future(x)
    else:
        future = Future()
        future.set_result(x)
        return future
```

Table 6: Output samples of Pretrained and Instruction-tuned models.