Evaluating Large Language Models in Class-Level Code Generation

Xueying Du*, Mingwei Liu*†, Kaixin Wang*, Hanlin Wang*, Junwei Liu*, Yixuan Chen*, Jiayi Feng*, Chaofeng Sha*, Xin Peng*, Yiling Lou*†
Fudan University
Shanghai, China

ABSTRACT

Recently, many large language models (LLMs) have been proposed, showing advanced proficiency in code generation. Meanwhile, many efforts have been dedicated to evaluating LLMs on code generation benchmarks such as HumanEval. Although being very helpful for comparing different LLMs, existing evaluation focuses on a *simple* code generation scenario (*i.e.,* function-level or statement-level code generation), which mainly asks LLMs to generate one single code unit (*e.g.,* a function or a statement) for the given natural language description. Such evaluation focuses on generating independent and often small-scale code units, thus leaving it unclear how LLMs perform in real-world software development scenarios.

To fill this knowledge gap, we make the first attempt to evaluate LLMs in a more *challenging* code generation scenario, *i.e.,* class-level code generation. Compared with existing code generation benchmarks, it better reflects real-world software development scenarios due to it comprising broader contextual dependencies and multiple, interdependent units of code. We first manually construct the first class-level code generation benchmark ClassEval of 100 class-level Python code generation tasks with approximately 500 person-hours. Based on the new benchmark ClassEval, we then perform the first study of 11 state-of-the-art LLMs on class-level code generation. Based on our results, we find that all LLMs perform much worse on class-level code generation compared to the method-level. While GPT models still dominate other LLMs on class-level code generation, the performance rankings of other models on method-level code generation no longer holds for class-level code generation. Besides, most models (except GPT models) perform better when generating the class method by method; and they have the limited ability of generating dependent code. Based on our findings, we call for software engineering (SE) researchers' expertise to build more LLM benchmarks based on practical and complicated software development scenarios.

*X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng and Y. Lou are with the School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, China.
†Corresponding authors (liumingwei@fudan.edu.cn; yilinglou@fudan.edu.cn)

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3639219

CCS CONCEPTS

• **Software and its engineering** → *Software evolution*; **Automatic programming**.

KEYWORDS

Class-level Code Generation, Large Language Model, Benchmark

ACM Reference Format:
Xueying Du*, Mingwei Liu*†, Kaixin Wang*, Hanlin Wang*, Junwei Liu*, Yixuan Chen*, Jiayi Feng*, Chaofeng Sha*, Xin Peng*, Yiling Lou. 2024. Evaluating Large Language Models in Class-Level Code Generation. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639219

1 INTRODUCTION

Code generation techniques automatically generate code snippets for the given natural language description, which can be leveraged to improve development productivity and have been extensively studied in literature [38, 39, 60]. The recent advance in large language models (LLMs) has brought significant advancements in the code generation domain. To date, researchers have proposed various LLMs [10, 13, 19, 27, 30, 42, 49, 53, 65, 75, 76] (such as GPT-4 [53], WizardCoder [49], and Instruct-CodeGen [10]) by training large models with over billions of parameters on massive general or code-specific corpora and instructions.

To fully understand the code generation capability of emerging LLMs, many efforts have been dedicated to evaluating LLMs on automatically or manually constructed code generation benchmarks. To date, many code generation benchmarks have been proposed, such as HumanEval [21] and MBPP [15]. Although being very helpful for people to understand and compare the performance of different LLMs, existing evaluation actually focuses on a rather *simple* code generation scenario, *i.e.,* function-level or statement-level code generation. They mainly ask LLMs to generate one single code unit (*e.g.,* a function or a statement) for the given natural language descriptions in a standalone way, which inherently have two limitations in evaluating LLMs in code generation. *First, such evaluation tends to focus on generating code of short length, e.g.,* each task in the most widely-used benchmark HumanEval only involves generating code of 11.5 lines and 24.4 tokens on average. Such a number of generated tokens is far within the maximum number of tokens in recent LLMs (*e.g.,* 2,048 for WizardCoder [49]). Therefore, it remains unclear about the further potential of LLMs in generating long code snippets. *Second, such evaluation mainly focuses on generating one single code unit, e.g., one function or one statement.* However, as shown in previous work [67], only 30% of methods are independent to other code contexts in the open-source projects.

Therefore, it remains unclear how LLMs perform in real-world software development scenarios, *i.e.,* generating a compound code unit of multiple methods[1] which are dependent on each other (*e.g.,* invoking each other or accessing the same variable).

Benchmark ClassEval. To fill this knowledge gap, this work makes the first attempt to evaluate LLMs in a more *challenging* code generation scenario, *i.e.,* class-level code generation. In particular, we evaluate the model capability of generating a class of multiple interdependent methods for the given natural language description. A class-level code generation benchmark stands out from more complex function-level benchmarks in two key ways: (1) ClassEval could not only assess correctness of generated code but also examines how well the model could incorporate contextual dependencies when generating a compound code unit; (2) ClassEval allows for further exploration different generation strategies (e.g., incremental or compositional method-by-method strategies to generate the whole class).

We manually construct the first class-level code generation benchmark ClassEval in a rigorous and time-intensive way, which takes approximately 500 person-hours to construct 100 class-level Python code generation tasks. Overall, ClassEval covers a wide range of topics in practical software development (*e.g.,* management systems and game development). Each task is constructed with a test suite of high testing sufficiency (*e.g.,* 98.2% and 99.7% branch-level or statement-level coverage) so as to facilitate reliable correctness checking of the generated code; furthermore, each task is designed to generate a class of multiple methods with diverse dependencies (*e.g.,* field, method, and library dependencies).

Empirical study. Based on the new benchmark ClassEval, we then perform the first study to evaluate LLMs on class-level code generation. In particular, our experiments include 11 state-of-the-art LLMs, which are diverse in model sizes, foundation models, sources, or domains. For each studied LLM, we explore its performance in generating class-level code with three different generation strategies, *i.e.,* holistic generation (generating the entire class all at once), incremental generation and compositional generation (generating the class method by method). For each generated code snippet, we measure its correctness with the widely-used metric *Pass@k* [21]. In addition, we also investigate the model ability of generating dependent code and analyze bad cases of incorrect classes.

Main findings and implications. Based on our results, we have the following main findings. First, we find that all existing LLMs show much worse performance on class-level code generation compared to on standalone method-level code generation benchmarks like HumanEval; and the method-level coding ability cannot equivalently reflect the class-level coding ability among LLMs. Second, we find that GPT-4 and GPT-3.5 still exhibit dominate superior than other LLMs on class-level code generation, and the second-tier models includes Instruct-StarCoder, Instruct-CodeGen, and WizardCoder with very similar performance. Third, we find that generating the entire class all at once (*i.e.,* holistic generation strategy) is the best generation strategy only for GPT-4 and GPT-3.5, while step-by-step generation (*i.e.,* incremental and compositional) is better strategies for the other models with limited ability of understanding long instructions and utilizing the middle information. Lastly, we find the limited model ability of generating method-dependent code and discuss the frequent error types in generated classes. Based on our findings, we summarize several practical implications, especially the appeal for SE researchers' expertise to build more LLM benchmarks of practical and complicated software development scenarios.

In summary, this paper makes the following contributions:

- **The first benchmark ClassEval** for class-level code generation, which is manually constructed with 500 person-hours and publicly available both on Github [1] and Hugging Face [2];
- **The first study** to evaluate 11 representative LLMs on class-level code generation with three different generation strategies;
- **Findings and implications** on analyzing the model capability and future directions for LLMs on class-level code generation.

2 BACKGROUND

We first introduce the recent LLMs for code generation in Section 2.1 and then motivate our study by revisiting existing code generation benchmarks in Section 2.2.

2.1 Large Language Models for Code Generation

Code generation is a task focusing on generating code snippets for the given natural language description, which has been extensively studied in recent literature [38, 39, 60]. *General LLMs*, which are large models with more than billions of parameters trained on general textual/code corpora and instructions, demonstrate remarkable capabilities not only in general NLP tasks [20] but also promising performance in code generation. For example, GPT-4 achieves the highest pass rate on HumanEval benchmark [49]. There has recently been an increasing trend to evaluate the code generation capacity even for general LLMs [21, 56]. *Code LLMs*, which are large models mainly trained with massive code-specific corpora and instructions, often have better capability than general LLMs in code generation tasks [24, 49, 70]. Existing code LLMs are designed with different training objectives. For example, some are using next-token prediction, while some code LLMs are trained with "filling-in-the middle" (FIM) capability, *i.e.,* infilling the missing portion based on the context. To date, a large number of code LLMs have been proposed [10, 11, 49].

2.2 Existing Benchmarks for Code Generation

Code generation benchmarks typically include various coding tasks where a natural language description serves as input, and the corresponding code serves as the ground truth output. Evaluation metrics such as passing rate (*Pass@k* [21]) are commonly used to assess the correctness of the generated code.

To date, many code generation benchmarks have been constructed via automated or manual manners. In this study, we revisit widely-used code generation benchmarks from the three following sources: (i) Top-10 popular datasets with the highest download volumes from Huggingface code generation datasets [3], (ii) benchmarks associated with recent LLM papers (released between June 2021 and June 2023), and (iii) enhanced benchmarks such as HumanEval+ [44] and Multi-HumanEval [14]. Table 1 provides an overview of the 13 distinct benchmarks collected from the three

[1]As we currently focus on Python, we distinguish concepts "*method*" and "*function*": a method is associated to an object and requires an object instance to be invoked, while a function is an independent code block that can be called from anywhere.

Table 1: Existing Benchmarks for Code Generation

Benchmark	Time	Language	Manual/Automated	Source	Granularity	#Tasks	#Tests	#LOC	#Tokens	Input Information
Concode [35]	2018	Java	Automated	Github	Function-level	2,000	-	-	26.3	NL
CoNaLA [66]	2018	Python	Automated	Stack Overflow	Statement-level	500	-	1	-	NL
APPS [32]	2021	Python	Automated	Contest Sites	Competitive	5,000	13.2	21.4	58	NL + Example Inputs/Outputs
HumanEval [21]	2021	Python	Manual	-	Function-level	164	7.7	11.5	24.4	NL + Function Signature + Example Inputs/Outputs
MBPP [15]	2021	Python	Manual	-	Function-level	974	3.0	6.8	24.2	NL
math-qa [15]	2021	Python	Manual	Math Study Sites	Statement-level	2,985	-	7.6	24.6	NL
Multi-HumanEval [14]	2022	Multilingual	Manual	-	Function-level	164	7.7	11.5	24.4	NL + Function Signature + Example Inputs/Outputs
MBXP [14]	2022	Multilingual	Manual	-	Function-level	974	3.0	6.8	24.2	NL
multi-math-qa [14]	2022	Multilingual	Manual	Math Study Sites	Statement-level	2,985	-	7.6	24.6	NL
CodeContests [43]	2022	Python, C++	Automated	Contest Sites	Competitive	165	203.7	59.8	184.8	NL + Example Inputs/Outputs
DS-1000 [40]	2022	Python	Automated	Stack Overflow	Statement-level	1,000	1.6	3.8	12.8	NL
HumanEval+ [44]	2023	Python	Manual	-	Function-level	164	774.8	11.5	24.4	NL + Function Signature + Example Inputs/Outputs
CoderEval [67]	2023	Python, Java	Automated	Github	Function-level	230	-	30	108.2	NL + Function Signature
ClassEval	**2023**	**Python**	**Manual**	-	**Class-level**	**100**	**33.1**	**45.7**	**123.7**	**Class Skeleton**

sources, including their release time, construction method (*i.e.,* manually written or automatically collected from public code corpus or competitions), benchmark size (#Tasks), target code granularity, target code language, code scale (#LOC: average lines of code, #Tokens: average number of tokens), average number of test cases per task (#Tests), and detailed input information (NL indicates natural language description information). We also present our constructed benchmark ClassEval in the last row for comparison.

Based on Table 1, we find that existing benchmarks actually shape a rather *simple* code generation scenario, which mainly evaluate the capability of LLMs in generating one single code unit (a function or a statement) in a rather *standalone* way. In particular, existing benchmarks typically focus on function-level or statement-level code generation tasks (Column "Granularity") and rarely include additional code contexts in the input (Column "Input Information"), which assumes that the code to be generated is an independent unit and thus leads to two limitations in evaluating LLMs.

First, existing benchmarks mainly focus on short code generation tasks, like generating one function or one statement. These tasks typically involve limited number of lines (*e.g.,* 1 to 30) and tokens (*e.g.,* 4.6 to 108.2), which may not fully explore the capabilities of recent LLMs that can handle much longer sequences, such as WizardCoder with 2,048 tokens. Thus, the potential of LLMs in generating longer code snippets remains unclear. Second, existing benchmarks mainly focus on generating independent code units without considering other code contexts. For instance, as shown in Figure 1, benchmarks like MBPP and HumanEval only provide limited information as input, such as natural language descriptions or function signatures with example inputs and outputs. However, in real-world scenarios, methods often depend on each other or share variables. Previous work [67] indicates that only 30% of methods in open-source projects are independent of other code contexts.While some recent code completion efforts [31, 74] focus on handling longer code contexts with more dependencies, the generated output still remains at the function or statement level. Therefore, it remains unclear how LLMs perform in generating a compound code unit of multiple methods which are dependent on each other (e.g., invoking each other or accessing the same variable).

Our Motivation. Existing benchmarks cannot facilitate the model evaluation on more complicated code generation tasks, such as generating longer and compound code units of multiple interdependent methods. To address this gap, we manually construct the first class-level code generation benchmark ClassEval and perform the first study to evaluate LLMs on class-level code generation

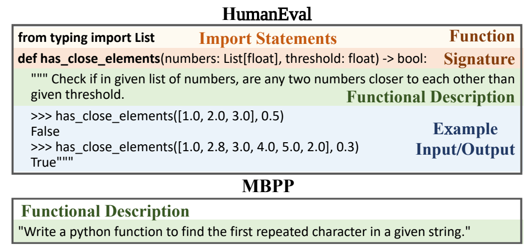


Figure 1: Examples in Existing Benchmarks

tasks, which ask LLMs to generate a class of multiple interdependent methods based on a given natural language description.

3 NEW BENCHMARK CLASSEVAL

In this section, we introduce our new benchmark ClassEval. We present the benchmark format (Section 3.1), the construction procedure (Section 3.2), and the benchmark characteristics (Section 3.3).

3.1 Benchmark Format

Each coding task in ClassEval comprises an input description for the target class (*i.e.,* the class to be generated), a test suite for verifying the correctness of the generated code, and a *canonical solution* that acts as a reference implementation of the target class.

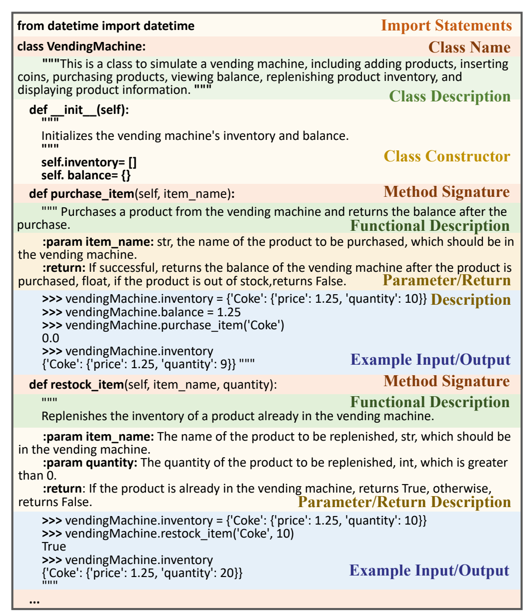


Figure 2: An Example of Class Skeleton in ClassEval

Typically, LLMs generate code snippets based on input descriptions and the correctness is verified with the provided test suite. The generated code must conform to a consistent interface (*e.g.,* the types of input parameters and return values) specified in the test suite for valid execution. For example, the benchmark HumanEval specifies the signature of the target function (Figure 1) to ensure that the generated bodies are validly checked by the given test suite. To achieve this, we define a ***class skeleton*** format for the input descriptions in our coding tasks. The class skeleton, inspired by contract programming [50], serves as a structured blueprint for the target class, containing both class-level information and method-level information. The class skeleton describes the functionalities of each method [64] and provides formal and precise specifications [45] for code generation by outlining expected behaviors, pre-conditions, and post-conditions. LLMs generate class-level code that aligns with the given test suite based on the class skeleton. The detailed definitions of elements in the class skeleton are in Table 2. Column "Mand." indicates whether the element is mandatory in the class skeleton. Figure 2 further illustrates an example of a class skeleton.

3.2 Benchmark Construction Procedure

Figure 3 illustrates the procedure of constructing ClassEval. We follow four steps to create ClassEval: (i) select suitable coding tasks using different strategies (Section 3.2.1); (ii) construct class skeletons based on the principles of contract programming [50] and test-driven development [18] (Section 3.2.2); (iii) create the test suite for each class skeleton (Section 3.2.3); and (iv) write the canonical solution for each coding task (Section 3.2.4). The constructed class skeletons, test suites, and canonical solutions form our class-level code generation benchmark ClassEval.

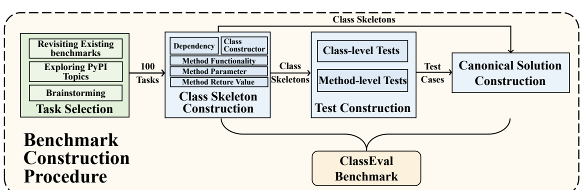


Figure 3: Overview of ClassEval Construction Process

To avoid the coding tasks being seen by LLMs during their training, our benchmark is constructed *completely manually*, so as to mitigate potential data leakages from existing code sources. Our manual construction involves a time-intensive process with approximately 500 person-hours on constructing 100 class-level coding tasks. Due to the significant manual efforts required, we currently stop the benchmark scale to this size. Moreover, following the trend of most existing benchmarks [15, 21], our benchmark primarily focuses on Python given its prevalence [57].

3.2.1 Task Selection. In this step, we design class-level coding tasks (*i.e.,* a unique *class description* for each task as defined in Table 2) for our benchmark.

Inclusion Sources. We design our coding tasks to cover diverse and real-world development topics, based on the following three sources. (i)*Revisiting Existing Benchmarks.* We refer to well established benchmarks like HumanEval and MBPP (Table 1) to include prevalent and common topics, including Mathematical Operation (e.g., area calculations) and Data Formatting (e.g., binary conversions and time conversions). (ii) *Exploring PyPI Topics.* We manually explore the Python Package Index (PyPI) [7], which hosts a vast repository of Python software packages and provides a diverse range of potential task topics. These include File Handling (e.g., JSON file processors and CSV file processors) and foundational Natural Language Processing tasks (e.g., stop word removal). (iii) *Brainstorming.* By reviewing the software development projects on GitHub repositories and a further brainstorming session, we collect more tasks that origin from real-world development scenarios. These tasks are intricate enough to include rich class dependencies, but not excessively complex to exceed the model's capabilities, including Management Systems (e.g., student registration system and movie booking system), Game Development (e.g., Minesweeper game and Gomoku game), and Database Operations (e.g., library database operations and SQL query generation).

Exclusion Criteria. Our benchmark focuses on coding tasks that can be implemented within one single class. Therefore, we exclude tasks that have complicated dependencies on the execution environment, including those related to (i) Network Programming, (ii) Graphical User Interface Design, (iii) Data Visualization, (iv) System Programming, and (v) Concurrent Programming. These tasks often require interactions with other classes or cannot be easily verified with assertion statements in unit tests.

In this way, we obtain a list of 100 diverse class-level coding tasks, covering a wide spectrum of topics, such as Game Development, File Handling, and Management Systems. Table 3 presents the topic distribution of our tasks.

3.2.2 Class Skeleton Construction. During this step, we manually construct the class skeleton for each coding task, involving 5 participants with an average of 3 years of Python development experience. Among these participants, one individual serves as the lead, responsible for final review and arbitration, while the remaining four are divided into two pairs. Each pair is tasked with creating 50 class skeletons, with one member responsible for writing the class skeleton and the other for double-checking it. In case of disagreements, the lead facilitates discussions to reach a consensus on the class skeleton, adhering to the design principles. This procedure also served as an iterative refinement process for our class skeleton design principles. Initially, we operated with a rudimentary set of principles, only outlining the fundamental elements of a class skeleton and a foundational principle on dependency. As more instances emerged in the construction process, coupled with discussions to reconcile differences and feedback from subsequent test and canonical solution construction phases, our class skeleton design principles progressively refined and enhanced. Ultimately, our comprehensive design principles are as follows.

Principle 1 (dependency): Each class skeleton should contain methods with diverse dependencies, *i.e.,* the methods are dependent to other code contexts within the class. Previous work [67] has shown that the majority of methods (over 70%) are dependent on other code contexts in the project. Unlike previous benchmarks that focus on standalone function-level code generation, our class-level benchmark aims to capture the real-world scenario where

Table 2: Elements Defined in Class Skeleton

Elements		Mand.	Definition
Class Level Info.	Class Name	✓	The name of the target class
	Class Description	✓	The description of the overall functionality of the target class
	Import Statements	×	Indicating the external libraries or modules necessary for implementing the target class
	Class Constructor	×	The initial method automatically invoked to initialize the attributes once the class is instantiated
Method Contract Design	Method Signature	✓	Defining the target method name, input parameters, and return type
	Functional Description	✓	Natural language descriptions on the functionality of each method
	Parameter/Return Description	×	Textual descriptions on expected inputs (*e.g.,* parameter types) and outputs (*e.g.,* return values) for each method
	Example Input/Output	×	Concrete examples of input values and corresponding output values on executing the target method

Table 3: Topic Type Definitions in ClassEval

Topic	Description	Examples	#Tasks
Management Systems	Operational functionalities in common software management systems projects	Student Registration System, Movie Booking System	27
Data Formatting	Processing data according to specific rules or patterns	Text-to-number Conversion, URL Format Validation	26
Mathematical Operations	Algorithms for mathematical and statistical problems	Basic Arithmetic Operations, Area Calculation	16
Game Development	Algorithms for game functionalities, including mechanics and state management	Minesweeper Game, Gomoku Game	10
File Handling	Common file operations including reading, writing, and simple processing data in files	CSV File Processor, JSON File Processor	9
Database Operations	Implementation of common database operations	Library Database Operation, SQL Query Generator	7
Natural Language Processing	Techniques for processing and analyzing text data	Stop Word Removal, Longest Word Identification	5

methods often have dependencies with other code contexts. To distinguish our benchmark from function-level ones, we deliberately avoid tasks that generate a class with independent methods, which would essentially be a collection of individual method-level coding tasks. Instead, class skeletons in our benchmark includes methods with diverse dependencies, including (i) *Library Dependency*, where methods rely on external libraries; (ii) *Field Dependency*, where methods depend on class instance variables (fields); (iii) *Method Dependency*, where methods rely on other methods within the same class; and (iv) *Standalone*, where methods function independently without dependencies on fields, methods, or external libraries.

Principle 2 (class constructor): The class constructor (if has) in each class skeleton should define the class fields and their default values. The constructor also includes natural language descriptions of the class fields to provide a clear understanding of their meanings. Importantly, the constructor does not make calls to other methods within the class to preserve the independence and self-contained nature of the class initialization process.

Principle 3 (method functionality): We avoid including complex functionalities like closing database connections, which are not easily testable and verifiable. Additionally, we enhance code reusability and maintainability by breaking down common and repetitive functionalities into separate methods. This principle fosters potential interdependencies between methods, simulating a more interconnected and practical coding scenario.

Principle 4 (method parameter): The method parameters are limited to primitive data types, avoiding object-level parameters or loosely defined arguments like ***kwargs*. This principle not only enhances clarity in method invocation but also facilitates testing, making it easier to create unit tests and verify the functionality of individual methods in isolation.

Principle 5 (method return value): Methods should include return values whenever possible for testing. For indicating success or failure, they use Boolean return types for standardization instead of custom strings. Additionally, method designs may encompass evaluative conditions for input parameters and include exception handling mechanisms. Detailed specifications of exception types, message content, and triggering circumstances are provided to ensure comprehensive testing and validation of exception handling.

Each constructed class skeleton would contain mandatory elements (*i.e.,* the class description, the class name, the method signature, and the functional description) and optional elements (*i.e.,* import statements, class constructor, parameter/return descriptions and the example input/output).

3.2.3 Test Construction. In this step, we manually construct a test suite for each coding task based on its class skeleton. The participants who were responsible for creating the class skeleton now take on the task of writing the corresponding test suite. Similarly, one participant focuses on writing the unit test cases, while the other ensures the quality and correctness of the test cases.

The methods in each class skeleton are designed to have multiple dependent relationships, as mentioned in Principle 1 in Section 3.2.2. Therefore, participants are required to construct test cases at two levels: *method-level tests* and *class-level tests*, so as to fully test the correctness of the implemented methods when they are invoked individually or together. *Method-level tests* primarily check the correctness of each method under test by independently invoking it without invoking any other methods in the class. On the other hand, *class-level tests* mainly check the correctness of multiple methods under test by invoking them sequentially together. Method-level tests ensure that the correctness of each method under test is individually checked without being impacted by the incorrect implementation of other methods, while class-level tests evaluate the overall correctness of the class by considering its interactions. Figure 4 provides two examples of both method-level and class-level test cases constructed for the class skeleton in Figure 2. Additionally, we include examples of test cases from existing benchmarks HumanEval and MBPP to highlight the differences. The function-level tests in existing benchmarks are comparable to the method-level tests in ClassEval, but the major difference is that function-level tests in existing benchmarks only check the return values of the function under test while our method-level tests further check the fields of the class. As shown in Figure 4, when testing the `purchase_item` method, the method-level test in ClassEval not only verifies the return value but also evaluates the operations performed on the `inventory` field. Moreover, existing benchmarks lack class-level tests since they primarily focus on single-function generation.

We then introduce the main principles of constructing method-level tests and class-level tests, respectively. For method-level tests, participants are asked to create at least five test cases to cover diverse scenarios of each method under test. For class-level tests, participants are required to construct test cases with different combinations of methods under test, ensuring that each method is invoked at least once in the class-level tests. To simplify test construction, participants are required to use the existing unittest framework [8], which provides diverse assertion APIs and a set of Test Fixtures (*e.g.,* `setUp` and `tearDown` methods) for preparation and cleanup tasks before and after test execution. Additionally, all constructed test cases are limited to a five-second running time to prevent potential infinite loops in the generated code.

3.2.4 Canonical Solution Construction. In this step, we manually write the canonical solution for each coding task based on its constructed class skeleton and test cases. Four participants (each with 2 - 4 years of Python development experience) who were not involved in constructing the class skeletons and test cases are engaged in this step. Each coding task is assigned to two participants, with one responsible for writing the canonical solution and the other for double-checking it. Participants are required to execute the solutions with test cases to identify and fix any bugs.

3.3 Benchmark Characteristics

In this way, we manually build a new benchmark ClassEval of 100 class-level coding tasks. The detailed characteristics are as follows.

Scale. ClassEval consists of 100 classes and 412 methods. To facilitate a direct comparison with other code generation benchmarks, we include the statistical data of ClassEval in Table 1. The results reveal large differences in lines of code for ClassEval (45.7) compared to the two most widely used handwritten benchmarks, HumanEval and MBPP, with multipliers of 4.0 and 6.7 respectively. Additionally, we perform additional statistics on the average number of tokens in the entire docstring information (class skeleton) in ClassEval (259.3), surpassing HumanEval (67.7) and MBPP (14.5) by a factor of 3.8 and 17.9 respectively. These results demonstrate that the class-level code generation task in ClassEval presents higher complexities, involving longer code generation, as well as more detailed and sophisticated docstring information.

Test Sufficiency. Table 4 provides comprehensive coverage statistics, encompassing traditional coverage metrics and advanced mutation testing results for the test cases in our benchmark, compared to HumanEval and MBPP. We collect the statement-level and branch-level coverage of the test cases on the canonical solution code using the Python toolkit *coverage* [4], and gather mutation testing results using the mutmut [5]. Additionally, we provide the average number of method-level tests (#Tests/M) and average class-level tests (#Tests/C). As shown in Table 4, the test cases in ClassEval not only achieve substantially higher statement-level and branch-level coverage (both over 98%) compared to HumanEval and MBPP but also exhibit superior performance in mutation testing (83.7%). This indicates more extensive and strong code checking for the generated solutions in our benchmark, which is supported by the fact that ClassEval also includes a larger number of method-level and class-level tests on average.

Table 4: Test Coverage and Test Cases Statistics

Benchmark	Statement	Branch	Mutation	#Tests/M	#Tests/C
HumanEval	98.8%	83.2%	82.3%	7.7	-
MBPP	98.6%	76.4%	72.4%	3.0	-
ClassEval	99.7%	98.2%	83.7%	8.0	33.1

Dependency. ClassEval focuses on class-level code generation tasks, distinguishing it from previous benchmarks. Table 5 shows the distribution of dependency levels within methods across ClassEval and previous benchmarks, as explained in Section 3.1. Notably, Library, Field, and Method dependencies are not mutually exclusive, and some methods may have a combination of Field and Method dependencies. We classify methods with either Field or Method dependencies as class-level dependent methods, totaling 314 (76.2%) within ClassEval. This inclusion makes ClassEval a comprehensive benchmark, suitable for evaluating LLMs that must account for intricate class-level interactions and contextual dependencies.

Table 5: Comparative Distribution of Dependency Levels

Dependency	MBPP	HumanEval	ClassEval
Standalone	974 (100%)	157 (95.8%)	58 (14.1%)
Library	-	7 (4.2%)	89 (21.7%)
Field	-	-	269 (65.5%)
Method	-	-	107 (26.0%)

Overall, in comparison to previous manually-crafted code generation benchmarks, ClassEval contains complicated class-level coding tasks involving larger-scale code snippets, diverse dependencies, sufficient test cases, and a wider range of topics from practical software development.

4 EMPIRICAL STUDY

Using ClassEval, we evaluate existing LLMs on class-level code generation to answer the following research questions.

- **RQ1 (Overall Correctness):** how do LLMs perform on class-level code generation?
- **RQ2 (Generation Strategies):** how do different generation strategies perform for LLMs on class-level code generation?
- **RQ3 (Dependency Generation):** how do LLMs perform on generating code dependent to other contexts during class-level code generation?
- **RQ4 (Bad Case Analysis):** what are the common errors during class-level code generation?

Table 6: Studied LLMs

	Model	Base Model	Time	Size	IF	FIM
Code LLM	Instruct-CodeGen [10]	CodeGen-multi [52]	2022.3	16B	✓	✓
	WizardCoder [49]	StarCoder [42]	2023.6	15B	✓	✓
	Instruct-StarCoder [11]	StarCoder [42]	2023.5	15B	✓	✓
	CodeGeeX [76]	-	2023.3	13B	✗	✗
	InCoder [30]	Dense [9]	2022.4	6B	✗	✓
	PolyCoder [65]	GPT-2 [55]	2022.2	2.7B	✗	✗
	SantaCoder [13]	GPT-2 [55]	2023.1	1.1B	✗	✓
General LLM	Vicuna [75]	LLaMA [59]	2023.3	7B	✓	✓
	ChatGLM [27]	GLM [71]	2022.3	6B	✓	✓
	GPT-3.5 [53]	-	2022.11	-	✓	✓
	GPT-4 [53]	-	2023.3	-	✓	✓

4.1 Studied LLMs

We select the state-of-the-art LLMs that have been widely studied in recent code generation work [44, 49]. In particular, we focus on recent models released since 2022, and we exclude the small models (with less than 1B parameters) due to their limited efficacy or the large models (with more than 20B parameters) due to our resource limits. Table 6 presents the 11 LLMs studied in our experiments with their releasing time (Column "Time"), model sizes (Column "Size"), and base models. In addition, we also summarize the training characteristics of the studied models, including whether the model has been trained to possess the ability of "filling-in-the-middle" (FIM) and whether it possesses the instruction-following (IF) ability via instruction tuning. Both FIM and IF capabilities are essential

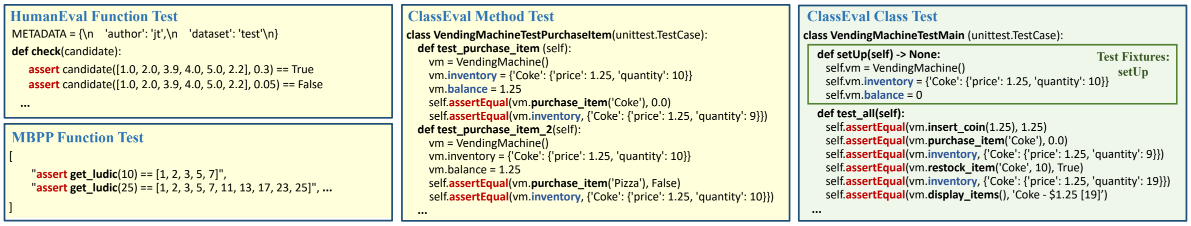


Figure 4: Test Cases in Existing Benchmarks and ClassEval

for the class-level code generation tasks. This is due to the requirement of these tasks to complete provided Class Skeletons, which encompass not only class-level information but also the design of all method contracts. Without FIM or IF capabilities, LLMs may only complete the final method solely on the final method contract for next-token prediction, thereby failing to generate a comprehensive class-level code. As shown in Table 6, our study includes a wide scope of LLMs that are diverse in multiple dimensions, such as (i) being both closed-source and open-source, (ii) utilizing different base models, (iii) covering a range of model sizes from 1B to 16B, (iv) being trained by both general or code-specific instructions, and (v) exhibiting different FIM and IF capabilities.

4.2 Studied Generation Strategies

Given a class-level code generation task, we study the performance of each model with three different generation strategies as follows:

- **Holistic Generation**: the model is asked to generate the entire class all at once with the class skeleton as inputs.
- **Incremental Generation**: the model is asked to generate the class in a method-by-method manner. Each iteration is based on the method bodies that have been generated in previous iterations. The iterative process repeats until all methods in the class are generated.
- **Compositional Generation**: the model is asked to generate the class in a method-by-method manner. Each iteration is independent, without considering the other generated methods. All the generated methods are assembled to form the class lastly.

The holistic generation strategy evaluates the model ability of handling long and complicated coding tasks all at once, while the incremental and compositional generation strategies focus on step-by-step class completion. The incremental strategy simulates progressive software development, where developers incrementally implement current methods based on existing ones. In constrast, the compositional strategy simulates real-world programming scenarios, where developers implement current methods based on other available method signatures. The compositional generation strategy is not influenced by the hints or the misleading information since it does not use other method implementation as input. Notably, both incremental and compositional generation strategies differ from standalone function-level code generation tasks in existing benchmarks like HumanEval, since our inputs include the class-level context such as the class constructor and other method signatures in the class skeleton.

4.3 Prompt Design

We then describe how we prompt LLMs to solve each class-level code generation task in ClassEval with each generation strategy.

LLMs with IF ability. Following the common practice of prompting LLMs with IF ability like WizardCoder [49], we set their prompts of two parts: (i) a *system prompt* as the beginning sentence to initialize the model, and followed by (ii) a *task instruction* to describe the goal of the task. Each generation strategy is set with its specific *task instruction*, *i.e., Instruction-H* for holistic generation, *Instruction-I* for incremental generation, and *Instruction-C* for a compositional generation. The prompt template is as follows.

> **System Prompt:** Provided below is an instruction detailing a task. Compose a response that aptly fulfills the request.

> **Instruction-H:** Please complete the class ${Class Name} in the subsequent code. ${Class Skeleton}

> **Instruction-I:** Please complete the method ${Method Name} within the following class ${Class Name}. ${Class-level Info} ${Generated Methods with Contract Designs} ${Target Method Contract Design}

> **Instruction-C:** Please complete the method ${Method Name} within the following class ${Class Name}. ${Class-level Info} ${Other Method Signatures} ${Target Method Contract Design}

LLMs without IF ability. The prompt of these models is the code context without any instruction: (i) for holistic generation, the prompt is just the class skeleton; (ii) for incremental generation, the prompt in each iteration includes the class-level information, generated methods, and the target method contract design; (iii) for compositional generation, the prompt for each method includes the class-level information, other method signatures, and the target method contract design.

4.4 Metrics

For correctness evaluation, we use the widely-used *Pass@k* [**?**] metric, which calculates the percentage of problems solved based on k code samples generated for each task:

$$\textbf{Pass@k} = \mathop{\mathbb{E}}_{\text{Problems}} \left[1 - \binom{n-c}{k} \Big/ \binom{n}{k} \right] \tag{1}$$

In Eq. 1, n represents the total number of samples, c denotes the number of correct samples, and k stands for k in *pass@k*. In particular, we calculate both class-level *Pass@k* and method-level *Pass@k* in class-level code generation tasks: class-level *Pass@k* considers code samples at the class granularity and method-level *Pass@k* consider code samples at the method granularity. A class-level code sample is deemed correct if it passes all the method-level and class-level test cases; and a method-level sample is deemed correct if it passes all the method-level test cases. In order to maintain an acceptable cost and response time in practical settings, we set n to five. To address the challenge of high sampling variance, we employ an unbiased estimator in line with previous work [21].

In addition to code correctness, we further measure the model capability of generating code that is dependent to the contexts (*i.e.,* invoking the other methods declared in the class or assessing the fields in the class). Such capability is essential in class-level code generation.s To this end, we design the metric *DEP*, which measures the recall of necessary and unique dependencies (Fields/Methods) in the canonical solution, indicating how many of these dependencies in the canonical solution are also used in the generated code. In particular, we consider method dependencies DEP(M) and field dependencies DEP(F):

$$\mathbf{DEP}(M) = \frac{\sum_{i=1}^{n} G_i(M)}{\sum_{i=1}^{n} S_i(M)} \quad (2) \qquad \mathbf{DEP}(F) = \frac{\sum_{i=1}^{n} G_i(F)}{\sum_{i=1}^{n} S_i(F)} \quad (3)$$

$G_i(M/F)$ is the number of generated method/field dependencies in the i^{th} method, and $S_i(M/F)$ is the number of actual method/field dependencies in the i^{th} method of the canonical solution. Note that if the same field or method is called multiple times, it's counted only once, ensuring that DEP(M) and DEP(F) values always fall within the range of [0,1].

For each generation strategy, we employ nucleus sampling to generate 5 samples and calculate *Pass@k* metrics with $k = \{1, 3, 5\}$. In addition, we also use the greedy sampling strategy to generate one single greedy sample and calculate *Pass@1* and *DEP* metrics. More sampling details are in Section 4.5.

4.5 Implementation Details

We use the OpenAI API interface, specifically the "gpt-4" and "gpt-3.5-turbo" model interface [6], in July 2023. For open-source LLMs, we directly obtain and run their released versions from their official repositories based on the documentation. The maximum window length is set to 2,048 tokens for all LLMs, determined by the smallest maximum window length among the studied LLMs.

In line with recent work [67], we consider two sampling methods for code generation: (i) nucleus sampling [33], where five solution code samples are randomly generated for each task with a temperature of 0.2 [21] and default top_p, and (ii) greedy sampling [22], where only one single solution code sample is generated for each task using greedy decoding, *i.e.,* setting the "do_sample" hyperparameter to false (temperature of 0). During each iteration in incremental and compositional generation, we obtain the Top-1 generated result for each method. Our experiments are run on a computational infrastructure comprising eight A800-80G GPUs.

Table 7: *Pass@k* with Nucleus Sampling on ClassEval

Model	Class-level			Method-level		
	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5
GPT-4	**37.6%**	**41.3%**	**42.0%**	**62.8%**	**67.4%**	**68.5%**
GPT-3.5	29.6%	34.9%	36.0%	50.4%	59.0%	61.1%
WizardCoder	12.2%	20.0%	23.0%	35.2%	47.1%	51.1%
Instruct-StarCoder	10.2%	12.7%	14.0%	23.1%	26.5%	27.7%
SantaCoder	8.6%	9.9%	10.0%	27.7%	33.0%	34.9%
Instruct-CodeGen	8.2%	12.3%	13.0%	24.9%	34.3%	37.1%
CodeGeeX	7.2%	9.4%	10.0%	21.2%	27.1%	29.5%
InCoder	6.2%	7.6%	8.0%	21.1%	26.5%	29.1%
Vicuna	3.0%	3.6%	4.0%	11.0%	15.8%	18.4%
ChatGLM	1.4%	2.6%	3.0%	8.2%	11.2%	12.4%
PolyCoder	1.4%	2.2%	3.0%	13.2%	17.5%	19.6%

5 RESULTS

5.1 RQ1: Overall Correctness

Figure 5 shows the class-level and method-level *Pass@1* with greedy sampling of studied LLMs on ClassEval and HumanEval. Due to space limits, we only present the best class-level *Pass@1* (and corresponding method-level *Pass@1*) for each model among the three generation strategies. A detailed comparison among three generation strategies is discussed in Section 5.3. Method-level *Pass@1* results on HumanEval are directly adopted from the latest work [49], and ChatGLM results on HumanEval are absent from existing evaluation. Table 7 presents the class-level and method-level *Pass@k* with nucleus sampling on ClassEval. Similarly, due to space limits, we only present results for the generation strategy with the highest class-level *Pass@1*. Based on Figure 5 and Table 7, we have the following observations.

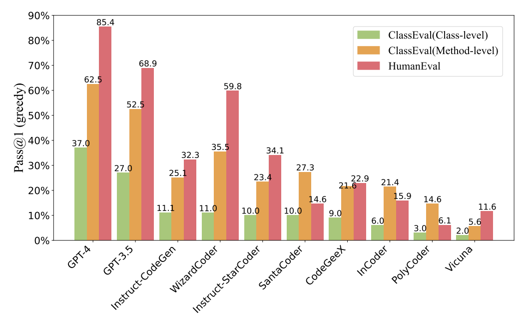


Figure 5: *Pass@1* (greedy) on ClassEval and HumanEval

Class-level code generation v.s. Method-level code generation. Based on Figure 5, we observe a significant decrease in correctness for all studied models on our class-level benchmark ClassEval compared to the existing method-level benchmark HumanEval. In particular, the best-performing models GPT-4 and GPT-3.5 achieve 85.4%/68.9% correctness on method-level tasks in HumanEval, but only 37.0%/27.0% correctness on class-level tasks in ClassEval. Similar trends can be observed on other models, *e.g.,* WizardCoder correctly generates 59.8% methods on HumanEval, but only 11.0% correct classes in our benchmark. Despite the inherent challenges of generating a class with multiple methods, the observed decrease in correctness on our benchmark ClassEval is not solely due to the larger number of methods to generate. The code generated by all models also shows lower method-level correctness on ClassEval compared to HumanEval. For instance, the method-level *Pass@1* of GPT-4 and GPT-3.5 drops from 85.4%/68.9% (on HumanEval) to 62.5%/52.5% (on ClassEval). This drop could be attributed to the complexity of generating code that depends on other context, which is known to be more challenging than generating standalone code. This finding is consistent with recent work [67]. In summary, our results show that existing LLMs still have limited performance in solving complicated coding tasks, such as class-level code generation.

We also observe that the model performance in the standalone method-level code generation tasks does not necessarily reflect their capability of class-level code generation. For example, while WizardCoder and Instruct-StarCoder exhibit much higher method-level *Pass@1* (59.8.4% and 34.1%) compared to SantaCoder (14.6%)

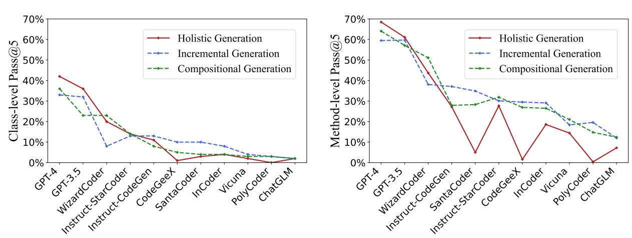
on HumanEval, all three model exhibit similar performance on class-level code generation tasks in ClassEval (around 10% - 11% *Pass@1*). This indicates that the method-level coding ability cannot equivalently represent the class-level coding ability among LLMs, further confirming the necessity of a class-level code generation benchmark.

Finding 1: Existing LLMs demonstrate substantially lower performance on class-level code generation tasks compared to standalone method-level code generation tasks. Additionally, the method-level coding ability cannot equivalently represent the class-level coding ability among LLMs. These findings strongly confirm the motivation and necessity of constructing class-level code generation benchmarks.

Comparison among models. As shown in Figure 5 and Table 7, GPT series (GPT-4 and GPT-3.5) substantially outperform all the other models on solving class-level coding tasks with both greedy sampling and nucleus sampling. For example, in Table 7, they outperform the third-ranked model WizardCoder by 25.4% and 17.4% in class-level *Pass@1* with nucleus sampling. Such results indicate the relatively stable dominance of GPT models when generalized to solve more challenging class-level coding tasks.

The second-ranked tier includes larger code models like Instruct-StarCoder, Instruct-CodeGen, and WizardCoder, achieving similar *Pass@1* with greedy sampling ranging from 10.0% - 11.1%. Notably, while these models show significant performance differences on method-level coding tasks in HumanEval, they perform similarly on class-level coding tasks. Smaller models (*e.g.,* PolyCoder) or general models (*e.g.,* ChatGLM) often exhibit worse performance, as expected due to the importance of model size and instruction datasets for generalization. The only exception is SantaCoder, which achieves comparable performance to larger code models with a much smaller model size.

Finding 2: On class-level code generation, GPT-4/GPT-3.5 still exhibits dominate superior than other LLMs; Instruct-StarCoder, Instruct-CodeGen, and WizardCoder perform similarly as the second tier; small or general models often perform the worse, except SantaCoder, which achieves comparable performance to larger models but with much less parameters.



(a) Class-level *Pass@5* (b) Method-level *Pass@5*

Figure 6: *Pass@5* of Three Generation Strategies

5.2 RQ2: Generation Strategies

Figure 6 compares the class-level *Pass@5* and method-level *Pass@5* of three different generation strategies. We find that the best generation strategy varies among different LLMs.

Holistic strategy v.s. others. On one hand, holistic generation is the best generation strategy only for the two models GPT-4 and GPT-3.5, which achieves much higher class-level *Pass@5* than the other two strategies (*i.e.,* the improvements range from 6% to 9% for GPT-4 and 4% to 14% for GPT-3.5). In addition, even for the method-level correctness, holistic generation still outperforms generating method in an incremental or compositional way (*i.e.,* 1.4% - 9.0% improvement in method-level *Pass@5*). On the other hand, the trends are different for the other models, which actually perform much better when generating the class method by method, namely with the incremental or compositional strategies. For example, in terms of the class-level correctness, CodeGeeX and SantaCoder generate 9% and 7% more correct classes with the incremental strategy compared to the holistic generation strategy. The main reason is that these models are able to generate much more correct methods (*i.e.,* 27.9% and 19.2% higher method-level *Pass@5*) when generating each method in separate iterations compared to generating all methods at once. Therefore, these models have higher chance to generate more correct classes if they are able to generate more correct methods with the incremental or compositional strategy.

One potential reason might be that most models (except GPT ones), exhibit rather limited capability of utilizing long input contexts, thus finding it more challenging to fully understand the code generation tasks given the entire class skeleton. As revealed by the recent work [47], LLMs often become substantially less effective with the increasing length of inputs; and in particular they tend to make better usage of the information located at the beginning or end of the inputs than that in the middle of inputs. Therefore, most existing LLMs perform better in generating a class method by method, since the task inputs are with the more atomic focus in such an incremental or compositional generation scenario; for models like GPT-3.5 and GPT-4 with a better understanding of long instructions, feeding the class-level context all at once is actually beneficial for them to fully capture and utilize the constraints between each method, leading to better class-level code correctness.

Incremental strategy v.s. compositional strategy. As for the two method-by-method strategies (*i.e.,* incremental and compositional strategies), we find the studied models actually have different preference on them. In particular, compared to the compositional generation manner, the additional inputs (the method body generated in previous iterations) in the incremental strategy are helpful for some models such as Instruct-CodeGen, InCoder, CodeGeeX, and SantaCoder. In contrast, the previously-generated method bodies can negatively affect the performance of models like Instruct-StarCoder and WizardCoder, resulting in a lower class-level correctness in incremental generation. In addition to the limited capability of handling long inputs mentioned above, another potential reason for the model's preference on a rather individual generation manner might be that the compositional generation aligns better with simple and atomic task instructions during instruction tuning.

Finding 3: Generating the entire class all at once (*i.e.,* holistic strategy) is the best generation strategy only for GPT-4 and GPT-3.5. For the other models, method-by-method generation (*i.e.,* incremental and compositional) works better. Such a disparity might stem from their limited capability of understanding the long instructions and utilizing the middle information.

5.3 RQ3: Dependency Generation

Method dependency v.s. Field dependency. Figure 7 presents the average field dependencies DEP(F) and the method dependencies DEP(M) of each model with the nucleus sampling. For space limits, we only present the best results among three generation strategies. Based on Figure 7, we can find that all models exhibit a much higher success rate in generating code dependent to fields than generating code dependent to other methods (*i.e.,* higher DEP(F) than DEP(M) on all the models). In other words, it might be much easier for models to generate field-accessing code than method-invoking code. In addition, among all the models, GPT models still show consistent superior in generating dependent code, *e.g.,* GPT-4 substantially outperform other LLMs by at least 12.6%/6.3% improvement in DEP(F)/DEP(M).

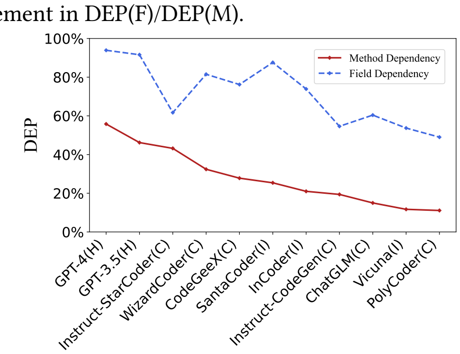


Figure 7: DEP(F) and DEP(M) in Nucleus Sampling

Impact of method dependency number. Given our observation above that it is more challenging to generate method dependency, we further investigate how each model performs at correctly generating code that invokes different number of other methods. Figure 8 is a stacked-bar plot that show the ratio of correctly-generated methods to all methods with the given number (*i.e.,* 0, 1, 2) of method dependencies (based on the canonical solution). Based on the figure, we can find that all the models perform best when generating methods that do not invoke any other method declared in the class (the blue bar in the figure). In addition, we find that no obvious difference when most models generate code invoking one other method (the green bar) or invoking two other methods (the yellow bar). In particular, for all the models, the average ratio of correctly-generated code that invokes one or two method(s) is 27.7% and 27.6% respectively.

Finding 4: It is easier for all the models to generate field-accessing code than method-invoking code. Additionally, they are better at generating standalone methods that do no invoke any other method.

5.4 RQ4: Bad Case Analysis

We further analyze the incorrectly-generated classes. To this end, we automatically parse the error logs generated during interpretation and execution, and present the error distribution of all models

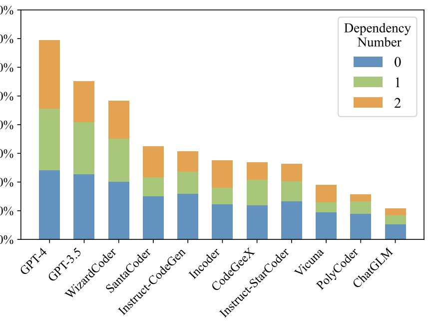


Figure 8: Distribution of correctly-generated methods in increasing method dependencies

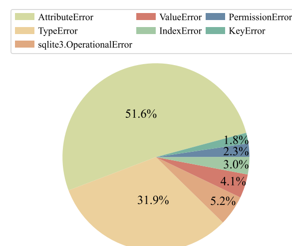


Figure 9: Error Distribution

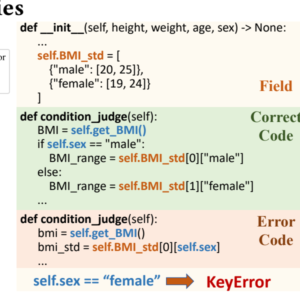


Figure 10: KeyError Code Example

in Figure 9. In particular, we find that most incorrect code encounters *AttributeError* and *TypeError*, indicating the limited model ability of understanding and satisfying syntactic or semantic constraints in the code context. Additionally, a few cases encounter *KeyError* due to erroneous operations on the dictionary variable. Figure 10 shows such an example from GPT-3.5, resulting from a misinterpretation of the field dependency. Specifically, the model erroneously accesses the first element of the field *BMI_std* list, which is a dictionary with the key "male". Attempting to access the key *self.sex* as "female" within this dictionary triggers a KeyError. This case indicate one of the challenges that LLMs might encounter in handling inherent class-level dependencies.

Finding 5: The classes generated by LLMs suffer from AttributeError and TypeError most frequently. In addition, the models might encounter difficulties in understanding the dependent contexts in the class.

6 IMPLICATION AND FUTURE DIRECTIONS

As the first class-level code generation benchmark, the major contribution of ClassEval is to reveal existing models' performance on class-level code generation for the first time and also to call for more future attention of improving LLMs on more challenging code generation. Based on our findings, we then discuss implications and explore how ClassEval can be utilized to improve the LLM as detailed below.

Prompting LLMs to be aware of dependencies can improve their performance in generating dependent code. As shown by our results, it is challenging for LLMs to generate dependent code. One potential prompt improving strategy is to address the model's attention to dependencies in the prompt. We further perform a preliminary experiment on GPT-3.5 by including the additional instruction "Please give special attention to the field and method dependencies" in the prompt and the results show that the enhanced prompt indeed improves the performance of LLMs with holistic generation strategy by increasing the accuracy of class-level Pass@1 from 26.0% to 29.0%.

Choosing the suitable generation strategy for LLMs can improve their performance in class-level code generation. As shown by our results, different models have different best generation strategies. For models with strong long-text-comprehending capabilities (e.g., GPT-4 and GPT-3.5), generating the whole class all at once (i.e., the holistic generation strategy) shows the best performance; for models with limited long-text-comprehending capabilities, generating the class method by method (i.e., incremental or compositional generation) is a better generation strategy. Therefore, one practical guideline for the future work is to focus on different class-level code generation strategies for different LLMs, e.g., designing novel step-by-step code generation strategies for LLMs such as CodeGeeX.

We then discuss the future work as follows.

Build more benchmarks for complicated and practical software development scenarios. Our results show a significant performance decrease of all studied LLMs on class-level code generation tasks, which are more complicated coding tasks derived from the practical software development scenario. In addition, the original performance rankings among most models and their original performance difference on function-level coding benchmarks no longer hold on class-level coding tasks. While this work makes the first attempt to construct class-level coding benchmarks for evaluating LLMs, there is still a large blank for benchmarks depicting practical software development tasks, *e.g.,* multi-class coding and domain-specific coding. Thus, we call for more efforts to construct such benchmarks for better understanding the model capability of solving practical and complicated development tasks and also for mitigating the overfitting phenomenon on existing benchmarks.

Enhance the model capability of understanding long instruction and solving compound tasks. Our results show that most models have limited capability of handling long instructions and utilizing the information in long contexts. Although there is a trend that LLMs can take longer and longer inputs [25], the effectiveness of utilization remains questionable. While researchers have proposed diverse prompting strategies such as chain-of-thought [29] and tree-of-thought [48] to improve model performance in solving complicated tasks, and this work also makes some initial explorations on three generation strategies for class-level coding tasks, we still call for more efforts in designing such strategies specifically for solving class-level coding tasks.

Improve the model capability of generating code dependent to the context, especially the method invocations. Our results show that all the models perform worse at generating code dependent on contexts, and correctly invoking methods is more challenging than accessing fields. Thus, we call for more efforts on improving model capability of understanding the constraints implied in code contexts by better prompting or tuning.

7 THREATS TO VALIDITY

Threats in benchmark construction. One potential threat is the data leakage between our benchmark and model training data, thus we *manually* construct the benchmark ClassEval. We also involve multiple participants to mitigate the subjectiveness and mistakes in manual participation. Another threat lies in the limited size and programming languages in our current benchmark, which cannot guarantee the generalizability of our findings, and we plan to continually extend our benchmark in the future.

Threats in empirical study. To avoid buggy model implementation, we adopt the public versions following official guidelines of each model. Another threat lies in the prompts used in our experiments, which might impact our findings. To avoid underestimating studied models, we perform a pilot study on a small set of prompt candidates and select the one with the best performance on three separate class-level coding tasks. We also report the results with greedy decoding, which is deterministic, so as to mitigate the randomness in model responses.

8 RELATED WORK

Since we have discussed most relevant work on code generation benchmarks in Section 2, we mainly introduce related work on LLMs for software engineering and LLM evaluation in this section.

LLMs for Software Engineering. In the field of software engineering (SE), LLMs have shown remarkable potential by being applied to an array of tasks. These include code generation [46, 54, 62, 72], code summarization [12, 51] and various software maintenance tasks, including vulnerability detection [58, 61, 73], test generation [68, 69], and program repair [26, 36, 37, 63]. These broad SE application stems from their robust training on extensive code and text data, which enhances their capabilities in both linguistic understanding and code comprehension.

LLM evaluation. Multi-faceted evaluation for LLMs is crucial for understanding the model capabilities given the black-box nature of LLMs. To date, the evaluation for LLMs has covered a wide range [20], encompassing not only traditional NLP tasks (*e.g.,* sentiment analysis [17], question answering [16], and reasoning [19]) but also some specific downstream domains (*e.g.,* medicine [23], agent [34] and recommendation system [28]). Specifically in software engineering domain, current evaluation focuses primarily on code comprehension and generation tasks [15, 21, 41, 44]. Many code LLMs (*e.g.,* Codex [21] and PanGu-Coder2 [56]) are released along with its rigorous evaluation on HumanEval to demonstrate their capabilities on code generation. While these previous efforts do not take scenarios beyond function-level code generation into account, our work fills this gap by manually constructing the first class-level code generation benchmark for evaluating LLM on more complicated and practical software development tasks.

9 CONCLUSION

This work makes the first attempt to evaluate LLMs on class-level code generation. We first manually construct the first class-level code generation benchmark ClassEval and perform the first study of 11 state-of-the-art LLMs on class-level code generation. We find that all LLMs perform much worse on class-level code generation compared to the method-level. While GPT models still dominate other LLMs on class-level code generation, the ranking of model performance on method-level code generation no longer holds in the class-level code generation. Besides, most models (except

GPT models) perform better when generating the class method by method; and they have the limited ability of generating dependent code.

ACKNOWLEDGMENTS

This work was supported by the National Key R&D Program of China (2021ZD0112903) and the National Natural Science Foundation of China under Grant No. 62302099.

REFERENCES

[1] [n. d.]. *ClassEval on GitHub*. https://github.com/FudanSELab/ClassEval
[2] [n. d.]. *ClassEval on Hugging Face*. https://huggingface.co/datasets/FudanSELab/ClassEval
[3] [n. d.]. *Code generation datasets in Huggingface*. https://hf.co/datasets?other=code-generation
[4] [n. d.]. *Coverage Library*. https://pypi.org/project/coverage
[5] [n. d.]. *mutmut*. https://mutmut.readthedocs.io/en/latest/
[6] [n. d.]. *OpenAI API interface*. https://platform.openai.com/docs/api-reference
[7] [n. d.]. *PyPI*. https://pypi.org/search
[8] [n. d.]. *Unittest Framework*. https://pypi.org/project/unitest
[9] 2021. *Dense-6.7B*. https://huggingface.co/KoboldAI/fairseq-dense-6.7B-Shinen
[10] 2023. *Instruct-CodeGen*. https://huggingface.co/sahil2801/instruct-codegen-16B
[11] 2023. *Instruct-StarCoder*. https://huggingface.co/GeorgiaTechResearchInstitute/starcoder-gpteacher-code-instruct
[12] Toufique Ahmed and Premkumar Devanbu. 2023. Few-Shot Training LLMs for Project-Specific Code-Summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 177, 5 pages. https://doi.org/10.1145/3551349.3559555
[13] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Muñoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, Joel Lamy-Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, Francesco De Toni, Bernardo García del Río, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luis Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. 2023. SantaCoder: don't reach for the stars! *CoRR* abs/2301.03988 (2023). https://doi.org/10.48550/arXiv.2301.03988 arXiv:2301.03988
[14] Ben Athiwaratkun, Sanjay Krishna Gouda, and Zijian Wang et al. 2023. Multi-lingual Evaluation of Code Generation Models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/pdf?id=Bo7eeXm6An8
[15] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *CoRR* abs/2108.07732 (2021). arXiv:2108.07732 https://arxiv.org/abs/2108.07732
[16] Yushi Bai, Jiahao Ying, Yixin Cao, Xin Lv, Yuze He, Xiaozhi Wang, Jifan Yu, Kaisheng Zeng, Yijia Xiao, Haozhe Lyu, Jiayin Zhang, Juanzi Li, and Lei Hou. 2023. Benchmarking Foundation Models with Language-Model-as-an-Examiner. *CoRR* abs/2306.04181 (2023). https://doi.org/10.48550/arXiv.2306.04181 arXiv:2306.04181
[17] Yejin Bang, Samuel Cahyawijaya, Nayeon Lee, Wenliang Dai, Dan Su, Bryan Wilie, Holy Lovenia, Ziwei Ji, Tiezheng Yu, Willy Chung, Quyet V. Do, Yan Xu, and Pascale Fung. 2023. A Multitask, Multilingual, Multimodal Evaluation of ChatGPT on Reasoning, Hallucination, and Interactivity. *CoRR* abs/2302.04023 (2023). https://doi.org/10.48550/arXiv.2302.04023 arXiv:2302.04023
[18] Thirumalesh Bhat and Nachiappan Nagappan. 2006. Evaluating the efficacy of test-driven development: industrial case studies. In *2006 International Symposium on Empirical Software Engineering (ISESE 2006), September 21-22, 2006, Rio de Janeiro, Brazil*. ACM, 356–363. https://doi.org/10.1145/1159733.1159787
[19] Ning Bian, Xianpei Han, Le Sun, Hongyu Lin, Yaojie Lu, and Ben He. 2023. ChatGPT is a Knowledgeable but Inexperienced Solver: An Investigation of Commonsense Problem in Large Language Models. *CoRR* abs/2303.16421 (2023). https://doi.org/10.48550/arXiv.2303.16421 arXiv:2303.16421
[20] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Kaijie Zhu, Hao Chen, Linyi Yang, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, Wei Ye, Yue Zhang, Yi Chang, Philip S. Yu, Qiang Yang, and Xing Xie. 2023. A Survey on Evaluation of Large Language Models. *CoRR* abs/2307.03109 (2023). https://doi.org/10.48550/arXiv.2307.03109 arXiv:2307.03109
[21] Mark Chen, Jerry Tworek, and Heewoo Jun et al. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374
[22] Siheng Chen, Rohan Varma, Aliaksei Sandryhaila, and Jelena Kovacevic. 2015. Discrete Signal Processing on Graphs: Sampling Theory. *IEEE Trans. Signal Process.* 63, 24 (2015), 6510–6523. https://doi.org/10.1109/TSP.2015.2469645
[23] Joseph Chervenak, Harry Lieman, Miranda Blanco-Breindel, and Sangita Jindal. 2023. The promise and peril of using a large language model to obtain clinical information: ChatGPT performs strongly as a fertility counseling tool with limitations. *Fertility and Sterility* 120, 3, Part 2 (2023), 575–583. https://doi.org/10.1016/j.fertnstert.2023.05.151
[24] Fenia Christopoulou, Gerasimos Lampouras, Milan Gritta, Guchun Zhang, Yinpeng Guo, Zhongqi Li, Qi Zhang, Meng Xiao, Bo Shen, Lin Li, Hao Yu, Li Yan, Pingyi Zhou, Xin Wang, Yuchi Ma, Ignacio Iacobacci, Yasheng Wang, Guangtai Liang, Jiansheng Wei, Xin Jiang, Qianxiang Wang, and Qun Liu. 2022. PanGu-Coder: Program Synthesis with Function-Level Language Modeling. *CoRR* abs/2207.11280 (2022). https://doi.org/10.48550/arXiv.2207.11280 arXiv:2207.11280
[25] Jiayu Ding, Shuming Ma, Li Dong, Xingxing Zhang, Shaohan Huang, Wenhui Wang, Nanning Zheng, and Furu Wei. 2023. LongNet: Scaling Transformers to 1, 000, 000, 000 Tokens. *CoRR* abs/2307.02486 (2023). https://doi.org/10.48550/arXiv.2307.02486 arXiv:2307.02486
[26] Xueying Du, Mingwei Liu, Juntao Li, Hanlin Wang, Xin Peng, and Yiling Lou. 2023. Resolving Crash Bugs via Large Language Models: An Empirical Study. *CoRR* abs/2312.10448 (2023). arXiv:2312.10448 https://doi.org/10.48550/arXiv.2312.10448
[27] Zhengxiao Du, Yujie Qian, Xiao Liu, Ming Ding, Jiezhong Qiu, Zhilin Yang, and Jie Tang. 2022. GLM: General Language Model Pretraining with Autoregressive Blank Infilling. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*. Association for Computational Linguistics, 320–335. https://doi.org/10.18653/v1/2022.acl-long.26
[28] Wenqi Fan, Zihuai Zhao, Jiatong Li, Yunqing Liu, Xiaowei Mei, Yiqi Wang, Jiliang Tang, and Qing Li. 2023. Recommender Systems in the Era of Large Language Models (LLMs). *CoRR* abs/2307.02046 (2023). https://doi.org/10.48550/arXiv.2307.02046 arXiv:2307.02046
[29] Hao Fei, Bobo Li, Qian Liu, Lidong Bing, Fei Li, and Tat-Seng Chua. 2023. Reasoning Implicit Sentiment with Chain-of-Thought Prompting. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*. Association for Computational Linguistics, 1171–1182. https://aclanthology.org/2023.acl-short.101
[30] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.
[31] Daya Guo, Canwen Xu, Nan Duan, Jian Yin, and Julian J. McAuley. 2023. LongCoder: A Long-Range Pre-trained Language Model for Code Completion. 202 (2023), 12098–12107. https://proceedings.mlr.press/v202/guo23j.html
[32] 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. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*. https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c24cd76e1ce41366a4bbe8a49b02a028-Abstract-round2.html
[33] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The Curious Case of Neural Text Degeneration. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=rygGQyrFvH
[34] Shaohan Huang, Li Dong, Wenhui Wang, Yaru Hao, Saksham Singhal, Shuming Ma, Tengchao Lv, Lei Cui, Owais Khan Mohammed, Barun Patra, Qiang Liu, Kriti Aggarwal, Zewen Chi, Johan Bjorck, Vishrav Chaudhary, Subhojit Som, Xia Song, and Furu Wei. 2023. Language Is Not All You Need: Aligning Perception with Language Models. *CoRR* abs/2302.14045 (2023). https://doi.org/10.48550/arXiv.2302.14045 arXiv:2302.14045
[35] 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, Brussels, Belgium, October 31 - November 4, 2018*. Association for Computational Linguistics, 1643–1652. https://doi.org/10.18653/v1/d18-1192
[36] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. *CoRR* abs/2302.05020 (2023). https://doi.org/10.48550/arXiv.2302.05020 arXiv:2302.05020
[37] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. InferFix: End-to-End Program Repair with LLMs. (2023), 1646–1656. https://doi.org/10.1145/3611643.3613892
[38] Sungmin Kang, Bei Chen, Shin Yoo, and Jian-Guang Lou. 2023. Explainable Automated Debugging via Large Language Model-driven Scientific Debugging. *CoRR* abs/2304.02195 (2023). https://doi.org/10.48550/ARXIV.2304.02195 arXiv:2304.02195

[39] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2312–2323. https://doi.org/10.1109/ICSE48619.2023.00194

[40] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen-tau Yih, Daniel Fried, Sida I. Wang, and Tao Yu. 2022. DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation. *CoRR* abs/2211.11501 (2022). https://doi.org/10.48550/arXiv.2211.11501 arXiv:2211.11501

[41] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2023. Enabling Programming Thinking in Large Language Models Toward Code Generation. *CoRR* abs/2305.06599 (2023). https://doi.org/10.48550/arXiv.2305.06599 arXiv:2305.06599

[42] Raymond Li, Loubna Ben Allal, and Yangtian Zi et al. 2023. StarCoder: may the source be with you! *CoRR* abs/2305.06161 (2023). https://doi.org/10.48550/arXiv.2305.06161 arXiv:2305.06161

[43] Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-Level Code Generation with AlphaCode. *CoRR* abs/2203.07814 (2022). https://doi.org/10.48550/arXiv.2203.07814 arXiv:2203.07814

[44] 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. *CoRR* abs/2305.01210 (2023). https://doi.org/10.48550/arXiv.2305.01210 arXiv:2305.01210

[45] Mingwei Liu, Xin Peng, Andrian Marcus, Christoph Treude, Xuefang Bai, Gang Lyu, Jiazhan Xie, and Xiaoxin Zhang. 2021. Learning-based extraction of first-order logic representations of API directives. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*. ACM, 491–502. https://doi.org/10.1145/3468264.3468618

[46] Mingwei Liu, Tianyong Yang, Yiling Lou, Xueying Du, Ying Wang, and Xin Peng. 2023. CodeGen4Libs: A Two-Stage Approach for Library-Oriented Code Generation. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 434–445. https://doi.org/10.1109/ASE56229.2023.00159

[47] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023. Lost in the Middle: How Language Models Use Long Contexts. *CoRR* abs/2307.03172 (2023). https://doi.org/10.48550/arXiv.2307.03172 arXiv:2307.03172

[48] Jieyi Long. 2023. Large Language Model Guided Tree-of-Thought. *CoRR* abs/2305.08291 (2023). https://doi.org/10.48550/arXiv.2305.08291 arXiv:2305.08291

[49] 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. *CoRR* abs/2306.08568 (2023). https://doi.org/10.48550/arXiv.2306.08568 arXiv:2306.08568

[50] Bertrand Meyer. 1992. Applying "Design by Contract". *Computer* 25, 10 (1992), 40–51. https://doi.org/10.1109/2.161279

[51] Nam V. Nguyen, Kim Q. Tran, Jaehong Lee, and Hung Nguyen-Xuan. 2024. Nonlocal strain gradient-based isogeometric analysis of graphene platelets-reinforced functionally graded triply periodic minimal surface nanoplates. *Appl. Math. Comput.* 466 (2024), 128461. https://doi.org/10.1016/J.AMC.2023.128461

[52] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A Conversational Paradigm for Program Synthesis. *CoRR* abs/2203.13474 (2022). https://doi.org/10.48550/arXiv.2203.13474 arXiv:2203.13474

[53] OpenAI. 2023. GPT-4 Technical Report. *CoRR* abs/2303.08774 (2023). https://doi.org/10.48550/arXiv.2303.08774 arXiv:2303.08774

[54] Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. (2022). arXiv:2201.11227 [cs.LG]

[55] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[56] Bo Shen, Jiaxin Zhang, Taihong Chen, Daoguang Zan, Bing Geng, An Fu, Muhan Zeng, Ailun Yu, Jichuan Ji, Jingyang Zhao, Yuenan Guo, and Qianxiang Wang. 2023. PanGu-Coder2: Boosting Large Language Models for Code with Ranking Feedback. *CoRR* abs/2307.14936 (2023). https://doi.org/10.48550/ARXIV.2307.14936 arXiv:2307.14936

[57] KR Srinath. 2017. Python–the fastest growing programming language. *International Research Journal of Engineering and Technology* 4, 12 (2017), 354–357.

[58] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. 2023. When GPT Meets Program Analysis: Towards Intelligent Detection of Smart Contract Logic Vulnerabilities in GPTScan. *CoRR* abs/2308.03314 (2023). https://doi.org/10.48550/ARXIV.2308.03314 arXiv:2308.03314

[59] Hugo Touvron, Thibaut Lavril, and Gautier Izacard et al. 2023. LLaMA: Open and Efficient Foundation Language Models. *CoRR* abs/2302.13971 (2023). https://doi.org/10.48550/arXiv.2302.13971 arXiv:2302.13971

[60] Vasudev Vikram, Caroline Lemieux, and Rohan Padhye. 2023. Can Large Language Models Write Good Property-Based Tests? *CoRR* abs/2307.04346 (2023). https://doi.org/10.48550/ARXIV.2307.04346 arXiv:2307.04346

[61] Chong Wang, Jianan Liu, Xin Peng, Yang Liu, and Yiling Lou. 2023. Boosting Static Resource Leak Detection via LLM-based Resource-Oriented Intention Inference. *CoRR* abs/2311.04448 (2023). https://doi.org/10.48550/ARXIV.2311.04448 arXiv:2311.04448

[62] Chong Wang, Jian Zhang, Yebo Feng, Tianlin Li, Weisong Sun, Yang Liu, and Xin Peng. 2024. Teaching Code LLMs to Use Autocompletion Tools in Repository-Level Code Generation. arXiv:2401.06391 [cs.SE]

[63] Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational Automated Program Repair. *CoRR* abs/2301.13246 (2023). https://doi.org/10.48550/ARXIV.2301.13246 arXiv:2301.13246

[64] Wenkai Xie, Xin Peng, Mingwei Liu, Christoph Treude, Zhenchang Xing, Xiaoxin Zhang, and Wenyun Zhao. 2020. API method recommendation via explicit matching of functionality verb phrases. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. ACM, 1015–1026. https://doi.org/10.1145/3368089.3409731

[65] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *MAPS@PLDI 2022: 6th ACM SIGPLAN International Symposium on Machine Programming, San Diego, CA, USA, 13 June 2022*. ACM, 1–10. https://doi.org/10.1145/3520312.3534862

[66] 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, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*. ACM, 476–486. https://doi.org/10.1145/3196398.3196408

[67] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Tao Xie, and Qianxiang Wang. 2023. CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models. *CoRR* abs/2302.00288 (2023). https://doi.org/10.48550/arXiv.2302.00288 arXiv:2302.00288

[68] Shengcheng Yu, Chunrong Fang, Yuchen Ling, Chentian Wu, and Zhenyu Chen. 2023. LLM for Test Script Generation and Migration: Challenges, Capabilities, and Opportunities. *CoRR* abs/2309.13574 (2023). https://doi.org/10.48550/ARXIV.2309.13574 arXiv:2309.13574

[69] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation. *CoRR* abs/2305.04207 (2023). https://doi.org/10.48550/ARXIV.2305.04207 arXiv:2305.04207

[70] Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Wang Yongji, and Jian-Guang Lou. 2023. Large Language Models Meet NL2Code: A Survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Toronto, Canada, 7443–7464. https://aclanthology.org/2023.acl-long.411

[71] Aohan Zeng, Xiao Liu, and Zhengxiao Du et al. 2023. GLM-130B: An Open Bilingual Pre-trained Model. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/pdf?id=-Aw0rrrPUF

[72] Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 2022. An extensive study on pre-trained models for program understanding and generation. In *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*. ACM, 39–51. https://doi.org/10.1145/3533767.3534390

[73] Chenyuan Zhang, Hao Liu, Jiutian Zeng, Kejing Yang, Yuhong Li, and Hui Li. 2023. Prompt-Enhanced Software Vulnerability Detection Using ChatGPT. *CoRR* abs/2308.12697 (2023). https://doi.org/10.48550/ARXIV.2308.12697 arXiv:2308.12697

[74] 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. (2023), 2471–2484. https://aclanthology.org/2023.emnlp-main.151

[75] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. *CoRR* abs/2306.05685 (2023). https://doi.org/10.48550/arXiv.2306.05685 arXiv:2306.05685

[76] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Evaluations on HumanEval-X. *CoRR* abs/2303.17568 (2023). https://doi.org/10.48550/arXiv.2303.17568 arXiv:2303.17568