CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models

Hao Yu
School of Software and Microelectronics, Peking University
China
yh0315@pku.edu.cn

Bo Shen
Huawei Cloud Computing Technologies Co., Ltd.
China
shenbo21@huawei.com

Dezhi Ran
Key Lab of HCST (PKU), MOE; SCS
Peking University
China
dezhiran@pku.edu.cn

Jiaxin Zhang
Huawei Cloud Computing Technologies Co., Ltd.
China
zhangjiaxin35@huawei.com

Qi Zhang
Huawei Cloud Computing Technologies Co., Ltd.
China
zhangqi98@huawei.com

Yuchi Ma
Huawei Cloud Computing Technologies Co., Ltd.
China
mayuchi1@huawei.com

Guangtai Liang
Huawei Cloud Computing Technologies Co., Ltd.
China
liangguangtai@huawei.com

Ying Li*
School of Software and Microelectronics, Peking University
China
li.ying@pku.edu.cn

Qianxiang Wang*
Huawei Cloud Computing Technologies Co., Ltd.
China
wangqianxiang@huawei.com

Tao Xie*
Key Lab of HCST (PKU), MOE; SCS
Peking University
Beijing, China
taoxie@pku.edu.cn

ABSTRACT

Code generation models based on the pre-training and fine-tuning paradigm have been increasingly attempted by both academia and industry, resulting in well-known industrial models such as Codex, CodeGen, and PanGu-Coder. To evaluate the effectiveness of these models, multiple existing benchmarks (e.g., HumanEval and AiXBench) are proposed, including only cases of generating a standalone function, i.e., a function that may invoke or access only built-in functions and standard libraries. However, non-standalone functions, which typically are not included in the existing benchmarks, constitute more than 70% of the functions in popular open-source projects, and evaluating models' effectiveness on standalone functions cannot reflect these models' effectiveness on pragmatic code generation scenarios (i.e., code generation for real settings of open source or proprietary code).

To help bridge the preceding gap, in this paper, we propose a benchmark named CoderEval, consisting of 230 Python and 230 Java code generation tasks carefully curated from popular real-world open-source projects and a self-contained execution platform to automatically assess the functional correctness of generated code. CoderEval supports code generation tasks from six levels of context dependency, where context refers to code elements such as types, APIs, variables, and consts defined outside the function under generation but within the dependent third-party libraries, current class, file, or project. CoderEval can be used to evaluate the effectiveness of models in generating code beyond only standalone functions. By evaluating three state-of-the-art code generation models (CodeGen, PanGu-Coder, and ChatGPT) on CoderEval and HumanEval, we find that the effectiveness of these models in generating standalone functions is substantially higher than that in generating non-standalone functions. Our analysis highlights the current progress and pinpoints future directions to further improve a model's effectiveness by leveraging contextual information for pragmatic code generation.

*Corresponding author

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

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3623316

CCS CONCEPTS

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

KEYWORDS

Code Generation, Large Language Models, Benchmark

ACM Reference Format:
Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models. In *2024*

IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3623322

1 INTRODUCTION

Recent years have seen a trend to tackle open-domain code generation tasks with machine learning techniques, especially large generative pre-trained language models [3, 4, 20, 22] based on Transformer [26], such as Codex [7], AlphaCode [18], InCoder [9], CodeGen [21], PanGu-Coder [8], and ChatGPT [24]. Given natural language descriptions specifying the functionalities of the function under generation, these models can generate both standalone functions (i.e., functions that invoke or access only built-in functions and standard libraries) and non-standalone functions.

To fairly and comprehensively evaluate the effectiveness of the preceding models, representative benchmarks [2, 5–7, 10–12, 16, 27, 28] are required and widely used in the literature. Released alongside Codex [7], **HumanEval** is a benchmark for Python to assess the functional correctness of programs generated by code generation models. HumanEval consists of 164 hand-written problems, each of which includes a function signature, a docstring, a canonical reference function, and multiple unit tests. Recently, **DS-1000** [16] is proposed to evaluate the effectiveness of code generation models in generating code that relies on third-party data science libraries. In addition to Python, there are benchmarks for Java (**AixBench** [11]) and other programming languages (**MultiPL-E** [5]), facilitating the understanding and development of code generation models for these other languages.

Despite the importance and usefulness of these preceding benchmarks, there exists a gap between these benchmarks and pragmatic code generation scenarios (i.e., code generation for real settings of open source or proprietary code). On one hand, **standalone functions** are heavily focused by these benchmarks. In particular, all the preceding benchmarks except DS-1000 include only standalone functions. Note that DS-1000 includes standalone functions in addition to non-standalone functions that invoke API functions from only seven specific widely used third-party libraries in data science. On the other hand, **non-standalone functions** commonly exist in pragmatic code generation scenarios. After analyzing the 100 most popular projects written in Java and Python, respectively, on GitHub, we find that non-standalone functions account for more than 70% functions[1] of the open-source projects.

To help bridge the preceding gap, in this paper, we propose CoderEval, a **context-aware** benchmark, which can be used to evaluate code generation models on pragmatic code generation. According to the source of dependency outside the function under generation, we categorize code generation tasks into six levels (i.e., self_contained, slib_runnable, plib_runnable, class_runnable, file_runnable, and project_runnable), with details described in Section 3. Given that the existing benchmarks such as HumanEval cover only the first two levels, CoderEval can provide a more practical and representative evaluation of the effectiveness of code generation models in tasks of pragmatic code generation. Note that functions belonging to the first two levels correspond to standalone functions and the others correspond to non-standalone functions.

[1]Detailed statistics are listed in our open-source repository [1].

To construct CoderEval as a representative and diverse benchmark for tasks of pragmatic code generation, we select and curate code generation tasks from real-world projects in three steps. First, we select functions in open source projects from the most frequent 14 tags and with high star counts on GitHub. For each function[2], we extract the original docstring (i.e., the natural language description of the function), the function name and signature, the code implementation, and the corresponding test code (if exists) to form one function-level code generation task. Additionally, we analyze the detailed contextual information (where context refers to code elements such as types, APIs, variables, and consts defined outside the function under generation but within the dependent third-party libraries, current class, file, or project) through program dependence analysis and provide it as all-context (all accessible context) and oracle_context (the actually used context), so as to conduct a fine-grained evaluation about model effectiveness. Second, to mitigate data leakage (i.e., the original docstring has a high probability of being used as training data by large language models), we recruit 13 experienced engineers to provide a human-labeled version of description as the second docstring (i.e., human-labeled docstring) to complement the original docstring. Third, to improve the evaluation accuracy, we examine the test coverage of the existing tests provided by each project in CoderEval and manually write additional tests to achieve high test coverage. CoderEval currently supports Python and Java, with 230 functions from 43 Python projects and 230 functions from 10 Java projects.

To automatically evaluate code generation models, CoderEval calculates the Pass@k metric [7] automatically to assess the functional correctness of generated code. Since CoderEval involves functions with contextual dependency and non-primitive types, we build a project-level execution platform to provide a ready runtime environment that can automatically assess the functional correctness of generated code. We develop this platform based on Docker, where we clone and set up environments for all projects. Given multiple solutions generated by a model, the platform can automatically place the generated code in the proper location of the project under consideration, to simulate the scenario where a developer accepts the generated code in an actual IDE.

We conduct a comprehensive evaluation of three state-of-the-art code generation models (CodeGen [21], PanGu-Coder [8], and ChatGPT [24]) on CoderEval and compare the difference with HumanEval. In the evaluation, for each model, we analyze the overall and level-wise effectiveness, the ability to correctly utilize contextual information, and the effect of the two different natural language descriptions (i.e., the original and human-labeled docstrings). Furthermore, among these models, we compute the intersection and difference in terms of correctly solved tasks. Through the analysis and comparison, we provide a good understanding of existing models and shed light on future directions and further progress. Experimental results show that in CoderEval for Python and CoderEval for Java, the effectiveness of the three models in generating standalone functions is substantially higher than that in generating non-standalone functions. Considering that more than 70% of functions in open-source projects belong to non-standalone functions,

[2]In this paper, we use the term of function to refer to both Python function and Java method.

improving a model's ability to consider and use contextual information is vital for the practical value of this technology.

In summary, we make the following main contributions:

- We point out the limitation of the existing benchmarks through an analysis of the 100 most popular open-source projects written in Java and Python, respectively: the existing benchmarks such as HumanEval typically include only standalone functions, whereas non-standalone functions constitute more than 70% of functions in the open-source projects.
- We introduce CoderEval, a benchmark of pragmatic code generation. CoderEval originates from open-source projects from various domains and considers non-primitive types, third-party libraries, and project-specific contextual references. In addition, CoderEval includes the human-labeled docstring for the function under generation to complement the original docstring.
- We evaluate and compare three state-of-the-art code generation models (CodeGen, PanGu-Coder, and ChatGPT) on CoderEval. Experimental results indicate three important findings: (1) these models do not work as well on non-standalone functions as on standalone functions, (2) it is important yet challenging for all these models to generate code with contextual dependency, even for ChatGPT (the most powerful model), and (3) the choice of using the human-labeled docstring vs. the original docstring has an impact on code generation effectiveness.

CoderEval and all the experimental results are open-sourced [1] to continually evolve in the code generation community.

2 BACKGROUND

In this section, we first introduce existing large language models for code generation. Then, we introduce the benchmarks used by existing large language models.

2.1 Large Language Models for Code Generation

PanGu-Coder [8] is a pre-trained language model for the task of text-to-code generation, which is based on the PanGu-α architecture [29] and a two-stage training strategy. CodeGen [21] is a series of conversational text-to-code large language models trained on natural language corpora, corpora of multilingual code (i.e., code written in multiple programming languages), and datasets of Python code. Codex [7] is the first work to use large generative pre-trained models to generate complete functions from natural language. AlphaCode [18] specializes in programming contests and performs on par with average human developers. InCoder [9] is a unified generation model that can perform program synthesis (through left to right generation) and editing (through padding). InCoder is trained to generate code files from a large number of code bases with specific friendly licenses, where code regions are randomly masked and moved to the end of each file, allowing code to fill with bidirectional context.

2.2 Benchmarks for Code Generation

Our proposed CoderEval is so far the only benchmark that supports project-level code generation and uses the evaluation metrics with Pass@k, which can validate the functional correctness of the generated code.

Benchmarks that contain project-level functions are important and yet difficult to construct for two main reasons. First, it is necessary to ensure that the selected projects can be compiled and sandboxed to achieve successful execution of different projects. However, it is difficult to successfully build and run many existing open-source projects. Second, to speed up testing, for the function under generation, the often large number of its covering test cases from its belonging project needs to be desirably reduced but statically determining whether a test case covers the function under generation requires non-trivial construction of functional dependency diagrams and coverage stub analysis. In addition, when the function under generation is not covered by the test cases of its belonging project, the builders of the benchmarks need to have a deep understanding of the often complex logic of the belonging project before writing high-quality test cases.

Most existing benchmarks (e.g., HumanEval [7], MulitPL-E [5], DS-1000 [16], and AiXBench [11]) use Pass@k for validating the correctness of a generated function and they contain only standalone functions. Although DS-1000 can validate the correctness of generated non-standalone functions, these non-standalone functions are limited to those that invoke API functions from only seven specific widely used third-party libraries in data science.

Not being a benchmark, Concode [14] is a large dataset that contains over 100,000 functions from approximately 33,000 open-source Java repositories. In Concode, functions from the same repository are split into training, validation, and test sets, so that a deep learning model can be trained based on the training and validation sets, and then be tested on the test set. Each function included in Concode is in the form of a pair: its natural language annotation and code implementation.

Although Concode can support validating the effectiveness of code generation approaches on generating non-standalone functions, there are five main differences between CoderEval and Concode for highlighting CoderEval's advantages. (1) CoderEval validates the correctness of the generated function by executing it, whereas Concode does so based on the text similarity between the generated function and the ground-truth function. (2) CoderEval alleviates the problem of data leakage by providing human-labeled docstrings whereas docstrings in Concode are likely "seen" by large language models during the pre-training stage. (3) The functions in CoderEval have been carefully selected by developers, whereas the functions in Concode are automatically collected and have not been carefully selected by developers. (4) CoderEval supports both Java and Python languages, whereas Concode supports only Java. (5) CoderEval contains latest code from the past five years, whereas all functions in Concode come from open-source code before 2018.

3 CODEREVAL BENCHMARK

In this section, we introduce CoderEval, whose construction process is shown in Fig 1. The construction process of CoderEval includes three phases: dataset collection, dataset inspection, and evaluation process.

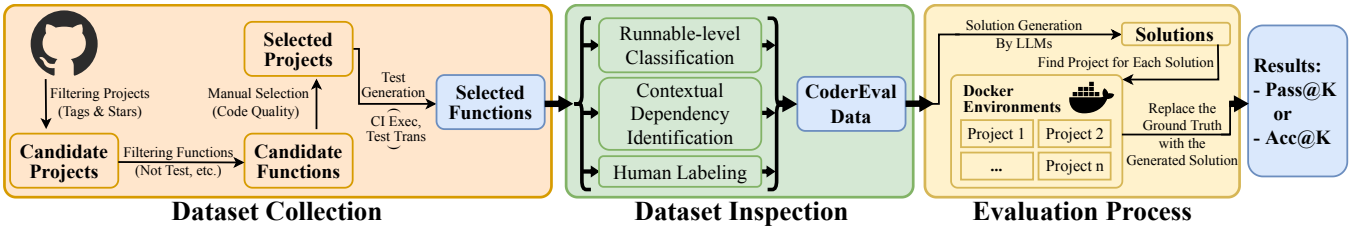


Figure 1: Overview of CoderEval construction process

3.1 Dataset Collection

3.1.1 Task Selection. To make CoderEval pragmatic and diverse, we select functions from various open-source projects by four steps. (1) We select candidate projects by crawling the tags of all projects on GitHub and selecting projects with the most frequent 14 tags and with high stars. For each tag, we select the projects with the top five highest number of stars. The 14 types of tags are "gson", "music", "logging", "chat", "websocket", "mvc", "leetcode", "microservices", "jdbc", "json", "crud", "datastructures", "log4j", and "serialization". (2) We extract all functions in the selected projects, and keep only the ones that are not a test, interface, or deprecated function, with a function-level comment in English, and can run successfully in the verification platform (Section 3.3) and pass the original test cases. (3) We select high-quality functions from the candidate functions through manual screening, whose main criterion is whether a function often appears in real development scenarios. (4) We attain projects according to the number of the selected functions contained in each project. This process can help us compile fewer projects with the same number of the selected functions. In the end, we attain 230 functions from 10 projects in Java. To maintain consistency with the number of functions in CoderEval for Java, we also attain 230 functions for CoderEval for Python from 43 projects.

The reason why we select high-quality functions manually (the third process in the previous paragraph) is that we want CoderEval to assess a model's ability to generate code that is helpful to developers. We have experienced developers select functions that may be used in real scenarios, following five rules. (1) Functions that contain fewer than ten contextual tokens. Excessive contextual dependencies make it difficult for a model to generate correct solutions. The functions that are too difficult for all large language models to generate correct implementations are not suitable for the benchmark. (2) Functions that are frequently used in real development scenarios judged by 13 developers. The rationale behind this rule is that different developers have various preferences over the frequently used functions; thus, the diversity of developers' preferences functions in CoderEval can eliminate potential bias. (3) Functions containing docstrings that can reflect the implementations of the functions. (4) Functions that have more than three lines of code implementation. (5) Functions that are not test or deprecated functions.

3.1.2 Test Construction. To improve the reliability of the evaluation, we need to do our best to make the test correct and complete. We analyze and get the unit tests contained in the projects, for the tests that have not achieved 100% line or branch coverage, the first author has manually written the additional test cases to achieve more high line and branch coverage for the functions in CoderEval. In CoderEval for Python, the input and output of test cases for a function obtained through CI are run in the entire project environment. Therefore, to ensure that the verification platform can correctly validate the function generated by the model, we execute test cases separately at the file level and use the output as the output of the test cases in the benchmark.

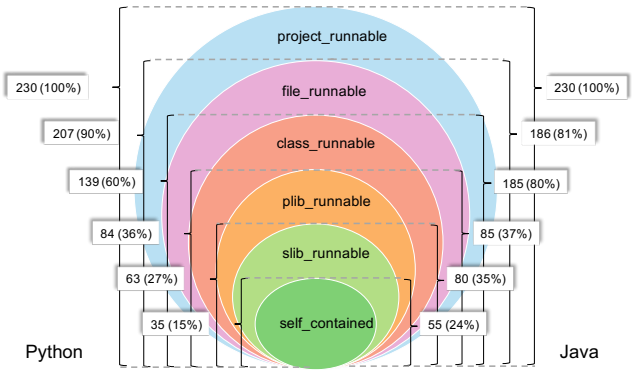


Figure 2: Distribution of runnable-levels in CoderEval

Unlike CoderEval for Python, the test cases in CoderEval for Java are provided by the unit test cases written in the original project or manually supplemented by authors. We select the unit test cases for CoderEval for Java with two steps. (1) To obtain the corresponding test cases of the function to be tested, we propose a function call analysis diagram based on source code to minimize the corresponding test cases for each function to be tested. The reason why we propose a function call analysis diagram is that there is no obvious relationship between the function under test with the test cases, when we need to assess the correctness of a function, we need to execute all test cases in the current project. However, it is time-consuming to execute all test cases of the current project for each function under test. (2) Since the execution of test cases is not only time-consuming but also depends on the test framework, we automatically convert test functions into non-test functions with the "main" function in a new Java file. The transaction process can help CoderEval not depend on the test framework. We assess the correctness of the function to be tested by calling the function through the "main" function. For this reason, we developed an algorithm to automatically convert test functions to non-test functions with a "main" function. For each function in CoderEval for Java, we add a new Java file with the "main" function. We denote these new Java files as "NewTestFile".

Table 1: Contextual Dependency Type Definition

Dependency Type	Definition	Examples in Python	Examples in Java
self-contained	built-in types/functions, no need to import	min(), print()	System.xxx
slib-runnable	standard libraries/modules, no need to install	os, subprocess, sys	Arrays.sort()
plib-runnable	publicly-available libraries on pypi/maven	unittest2, requests	com.google.code.gson
class-runnable	code outside the function but within class	self.xxx, X.f()	this.f()
file-runnable	code outside the class but within the file	func(), URL, name	func()
project-runnable	code in the other source files	superclass, utils	superclass, utils

Each level of runnable must rely on the dependencies defined at the current level and must not rely on the dependencies defined at subsequent levels. For its previous level of dependency, it can be dependent or not.

3.2 Dataset Inspection

3.2.1 Human-labeled Docstring. The prompt is important for large language models [15, 23, 25, 30, 31], when it comes to code generation, the task specification can have a great impact on the quality of generated code. During the research and case study, we find that the quality of original natural language descriptions from the selected open-source projects is rather unstable and inconsistent. In practice, the function-level comment can play multiple different roles at different abstract levels or granularities, like explaining the internal logic, introducing the external usage and behavior, declaring the effect and caution, etc.

To study the effect of different prompts and mitigate the memory effect of language models, we recruit 13 professional software engineers with at least 3 years experience of Python/Java, and let them provide a human-labeled version of the function description with double-checking. There are three main reasons for including human-labeled docstrings in CoderEval. (1) Including human-labeled docstrings is to mitigate the memory effect (i.e., original docstrings have a high probability of being seen by existing large language models in the pre-training stage) of large language models and explore the impact of prompts with human labeling on the quality of code generated by large language models. (2) Including human-labeled docstrings is to study how the model performs on different docstrings. (3) Including human-labeled docstrings is to provide high-quality docstrings for the functions in CoderEval. In particular, 13 software engineers with at least 3-year experience of Python/Java additionally provide a human-labeled version of docstring for each problem in CoderEval.

3.2.2 Contextual Dependency Identification. One of the major differences between HumanEval and CoderEval is that we take the contextual dependency into consideration. When generating function-level code, it is often the case that a token undefined error will lead to an error in the generation of the entire function. Therefore, providing the information actually called in the function can help CoderEval validate the model's awareness of context information in a fine-grained way.

We denote the context code elements that a function directly requires to run as its contextual dependency. Table 1 shows the definition and examples for each dependency type. We identify the contextual dependency of a function through program analysis of the whole project with two steps. (1) Before the analysis, we first build a knowledge base that helps distinguish the references of Python/Java builtins, and the imports of standard or public libraries from the project-specific code. To do this, we cache all built-in types/functions/variables/constants and standard library names for each Python version from 3.0.0 to 3.10.0 and Java version from 1.8 to 17, as well as all public-available libraries on pypi.org and *Maven* central repository. (2) With the knowledge base, given a function to analyze, we retrieve the database to get the source file that it came from, then parse it to get a list of types/functions/variables/constants definitions. For each dependency type, the criterion for setting it to true was whether the function *used* the code of this type. **Note that we also use the same contextual dependency identification approach to calculate the proportion of standalone and non-standalone functions in the 100 most popular projects written in Java and Python, respectively, on GitHub.** (3) We employ static program analysis technologies to identify all the external references and invocations whose definitions are outside the current function and classify them into three categories: type_reference, variable_reference, and API invocation. More specifically, type_reference refers the user-defined class or the standard type (e.g., "List", "subprocess", and "os"). Variable_reference refers to the user-defined variable and object. API invocation refers to the user-defined function and function in a standard or third-party library.

We also further split the contextual information into *oracle_context* information and *all_context* information. We denote the information that the original function body actually depends on as the *oracle_context* information. Oracle_context information can not only be used to evaluate the accuracy of the context information in the code generated by the model but also can be used as part of the input to the model in the reasoning stage, to validate whether prompting the model with the oracle information can improve the effectiveness. However, it is usually difficult for the model to know oracle information in advance, so CoderEval also provides the *all_context* information, that is, all types, variables, and APIs defined/imported in the current file, to evaluate whether the model can pick the correct contextual information to fulfill the task. The contextual dependency identification technique is effective not only on CoderEval but also on other datasets and benchmarks.

In this paper, we use *oracle_context* information to evaluate the accuracy of the context information in the code generated by large

language models. Note that we do not use the *all_context* information in our experiments, all experiments in this paper use only original docstring, human-labeled docstring, function name, and function signature. We hope the *all_context* information is valuable for the research community, and we will study the impact of *all_context* information on code generation tasks in our future work.

3.2.3 Runnable-level Classification. Based on the contextual dependency, we classify the code generation tasks into six runnable levels, that is, the scope that the function can run successfully. These levels include: *self-contained*, *slib-runnable*, *plib-runnable*, *class-runnable*, *file-runnable*, and *project-runnable*. Each level of runnable must rely on the dependencies defined at the current level and must not rely on the dependencies defined at subsequent levels. For its previous level of dependency, it can be dependent or not. For example, *plib-runnable* means the function depend on public libraries, so it can run successfully outside the class, current source file and project, as long as the required Python version is satisfied and the libraries are installed and imported.

The left part and right part of Fig 2 show the distribution of runnable levels on CoderEval for Python and Java, respectively. Most (84%) of the collected functions in CoderEval for Python are *file_runnable*, which indicates that the context information in the current file is necessary and critical for a code generation model to generate the function correctly. The *class-runnable* functions account for half (54%) of all functions, which is not surprising considering the prevalence of object-oriented programming in open-source and non-trivial projects. On the contrary, *slib-runnable* accounts only for 17% in CoderEval for Python but 100% in the Codex dataset and the HumanEval benchmark. This fact proves the bias and mismatching in the model optimization and evaluation process with HumanEval, especially in the pragmatic generation of code in real settings. Similar to CoderEval for Python, *slib-runnable* accounts only for 35% in CoderEval for Java.

3.3 Evaluation Process

As we need an evaluation platform to provide a ready runtime environment with automatic programs to execute and validate the code generated by code generation models, we select to base it on a Linux Docker image, which can provide a virtual and safe sandbox to enable easy duplication and prevent harmful execution.

3.3.1 Evaluation for Python. To evaluate generated Python code, we need to clone and set up environments for all 43 projects. To avoid Python and library version conflicts, under each repository's root directory, we first use *pyenv* to set the local Python version to the highest version specified in the CI configuration or document, then use *venv* to create an individual virtual environment for the current project. After that, we use *pip* to install all dependencies and trigger the original tests in the project to validate the runtime environment.

With the environment ready and given a model to evaluate, we write a program to automatically *inject* the generated code into the project, correctly invoke the function with test input, and compare the actual output with the expected output. Given *n* code snippets generated by the model, it then sequentially replaces the original function with each of them to *simulate* a scenario when the developer accepts the generated code and tests it in an actual IDE. After the replacement, the entry point code will be triggered, and the running output (e.g., return values, exceptions, errors, and console output) will be captured. If the function returns with no error or exception, the platform will compare the actual output with the expected output, otherwise, the unexpected termination will be treated as a test failure. After all code snippets for all tasks are tested, the results of all tasks will be used to compute the final Pass@k metric.

Table 2: Statistical comparison among CoderEval, HumanEval, and Concode

Statistical / Benchmark	Cyclomatic Complexity	Line of Code
CoderEval-Python	4.71	32.0
CoderEval-Java	3.10	10.2
HumanEval	3.62	7.8
Concode	1.43	4.8

3.3.2 Evaluation for Java. Similar to the evaluation platform for Python, we also clone the involved projects first. We ensure that all projects in the current version of CoderEval for Java can be executed with Java 17 version. Different from Python, Java programs need to be compiled before execution, so we write a program to automatically compile the test file (i.e., "NewTestFile") in advance, and then dynamically replace and recompile the file that the function to be tested belongs to.

With the environment ready and given functions generated by large language models to evaluate, the platform uses the "javac" command to compile changed files incrementally, and the "java" command to execute the bytecode of "NewTestFile". To compile or execute a modified Java file, both the "javac" and "java" commands use the parameter, "-cp", to import the class file and third-party library dependency of the given file. Given *n* methods generated by the model, the platform will also automatically replace the original function with each of them, and try to recompile the residing file via the "javac" command. If the compilation fails, the test will be treated as a failure. If there is no error or exception after compilation, the platform will invoke the "java" command to execute the "NewTestFile". The return value of the command directly indicates the behavioral correctness of the generation code.

3.4 Statistical comparison with different benchmarks

Table 2 shows the average value of cyclomatic complexity and average number of lines of code in CoderEval, HumanEval, and Concode. We find that the cyclomatic complexity of functions in CoderEval is similar to HumanEval, and the average number of lines of code in CoderEval is more than that in HumanEval.

4 EVALUATION SETUP

In this section, we describe the setup of our experiment with three models (CodeGen, PanGu-Coder, and ChatGPT) on two benchmarks (CoderEval and HumanEval) in terms of research questions,

Table 3: Overall effectiveness of three models on two benchmarks

Benchmark	Model	Python			Java		
		Pass@1	Pass@5	Pass@10	Pass@1	Pass@5	Pass@10
CoderEval	CodeGen[1]	9.48%	19.58%	23.48%	13.91%	27.34%	33.48%
	PanGu-Coder[2]	11.83%	20.93%	27.39%	25.43%	37.39%	43.04%
	ChatGPT[3]	21.04%	27.31%	30.00%	35.39%	42.77%	46.09%
HumanEval	CodeGen[1]	10.2%	19.79%	22.8%	5.78%	9.0%	9.45%
	PanGu-Coder[2]	13.42%	21.48%	22.73%	8.21%	15.88%	18.63%
	ChatGPT[3]	39.21%	64.09%	72.96%	38.21%	59.35%	67.23%

[1] We use the 350M CodeGen-Mono model with the default settings for Python. Since CodeGen does not have a monolingual version of Java, on CoderEval for Java, we use the CodeGen-Multi model instead.
[2] PanGu-Coder has two different models for Python and Java, we use the 300M models with the default settings for Python and Java.
[3] We use the "gpt-3.5-turbo" for ChatGPT in our experiments.

model settings, and the evaluation metric. To reduce the random of a single experiment, we run each experiment ten times and find that the standard deviation of Pass@k for all models is about 1%. To better show the evaluations (e.g., complementarity and intersection of different models and the effectiveness of the generated standalone functions and non-standalone functions of each model), in this section, we use the median group of experiments selected from 10 experiments according to the Pass@10 value. Our experimental results are open-source [1].

4.1 Research Questions

Our experiment answers the following research questions:

- **RQ1:** How do CodeGen, PanGu-Coder, and ChatGPT perform on CoderEval and HumanEval?
- **RQ2:** How do different models perform in correctly utilizing the contextual information?
- **RQ3:** How does the prompt affect the effectiveness of different models?

4.2 Model selection and settings

4.2.1 Model selection. When doing experiments, we focus on models that support both Java and Python language code generation. CodeGen, PanGu-Coder, and ChatGPT all support both Java and Python. Specifically, CodeGen is chosen because it has specialized models for the Python language (CodeGen-mono) and models that support multiple languages (CodeGen-multi). The reason for choosing PanGu-Coder is that PanGu-Coder has two models specifically designed for Python and Java. ChatGPT is chosen because it is the most effective and parameter-intensive multilingual model. Selecting these models is helpful for comparing the effectiveness of single-language generative models and multi-language generative models. The reason why we do not select CodeGeex is that CodeGeex only has a model of 13B size. Due to our limited computing resources, we are unable to complete the CodeGeex experiment at the time of the paper writing. We plan to evaluate the effectiveness of CodeGeex and other recent open-source models (e.g., StarCoder [17] and WizardCoder [19]) on CoderEval in future work.

4.2.2 Model settings. For PanGu-Coder, we use the 300M PanGu-Coder model with the default settings. For CodeGen, we use the 350M CodeGen-Mono model with the default settings for Python. Since CodeGen does not have a monolingual version of Java, on CoderEval for Java, we use the CodeGen-Multi model instead. For ChatGPT, we use the "gpt-3.5-turbo" in our experiments. ChatGPT's model parameter scale is much larger than the other two models.

In the inference phase, for CodeGen and PanGu-Coder, we set the max window length to 1024. We use nucleus sampling [13]: the number of samples is 10, and the temperature is 0.6, which means generating 10 solutions per problem. Note that we do not use the *all_context* information in our experiments, all experiments in this paper use only original docstring, human-labeled docstring, and function signature.

4.3 Evaluation Metric

Similar to HumanEval, we adapt the Pass@k metric to assess the behavior correctness of generated code snippets according to test cases. As we set n (the number of samples) to 10 and calculate Pass@k for k in 1, 5, and 10, to avoid the issue of high sampling variance, we use the unbiased estimator of Pass@k implemented by ChatGPT in HumanEval[3]. Note that although n was set to 200 in papers of PanGu-Coder and CodeGen to explore the potential of models, we set n to ten as what the Copilot plugin does since we consider sampling ten times is more feasible and reasonable at an acceptable cost and response time in practical settings. Besides Pass@k, we also propose Acc@k metric, to evaluate the effectiveness of three models in correctly utilizing contextual information. Section 5.2.1 details the definition of Acc@k.

5 EXPERIMENTS

In this section, we detail the experiment results and analysis for CodeGen, PanGu-Coder, and ChatGPT on CoderEval.

5.1 RQ1: How do CodeGen, PanGu-Coder, and ChatGPT perform on CoderEval and HumanEval?

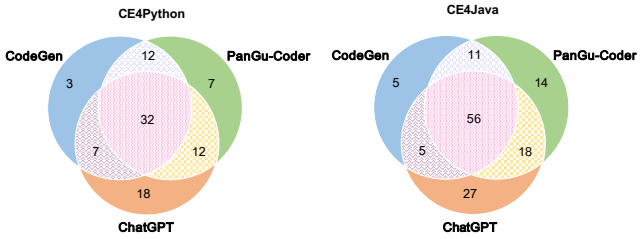
5.1.1 Overall effectiveness. Table 3 shows the overall effectiveness of CodeGen, PanGu-Coder, and ChatGPT on CoderEval and HumanEval. On HumanEval, the effects of the three models on Python

[3] https://github.com/openai/human-eval/

in HumanEval consist of the effects reported in their papers. Since the functions in CoderEval for Python and Java are different, we cannot directly compare the effectiveness of the three models on Python and Java in CoderEval. Both on HumanEval and CoderEval, ChatGPT consistently outperforms the other two models for Python and Java. The major reason is that ChatGPT's model size is much larger than the other two models we tested. CodeGen and PanGu-Coder perform worse on HumanEval than on CoderEval, because although the functions in HumanEval are standalone functions, their difficulty is higher than that of the standalone functions in CoderEval. The functions in HumanEval tend to be more algorithmic, while CoderEval tends to be more code based on real-world development scenarios. ChatGPT's performance in HumanEval is much better than that in CoderEval because of its powerful algorithmic learning ability, which makes it perform well in HumanEval. However, when generating code that relies on contextual information, it lacks sufficient contextual information, resulting in poor performance on CoderEval.

5.1.2 Complementarity and intersection of three models. Fig 3 shows the number of solved problems by the three models on CoderEval, with the difference and intersection. We use the median group of experiments selected from 10 experiments with the Pass@10 value to count the passing questions of each model. We consider a problem to be solved if the model can generate *at least* one correct solution that passes all tests out of the ten generated solutions. Among the 230 problems in CoderEval for Python, the total number of problems solved by the three models is 91 (complementarity), while 32 problems can be solved by any of the three models (intersection). On CoderEval for Java, the total number of problems solved by the three models is 136 (complementarity), while 56 problems can be solved by any of the three models (intersection). The high complementarity indicates that different models have their unique capabilities, and the high intersection indicates that the problem-solving ability of the three models is consistent on some tasks. We manually analyze the intersection tasks and find that more than half of them belong to standalone functions (i.e., *self-contained* and *slib-depend* levels), more specifically the total intersection numbers of CoderEval for Python and Java is 88, of which 52 are standalone functions. Note that the standalone functions only for about 30% in CoderEval. Therefore, the three models are all good at generating functions without external dependency (while ChatGPT performs particularly well), extending the ability to other runnable levels and exploring how to combine the code generation capabilities of different models is a worthwhile research direction.

5.1.3 Effectiveness comparison between the generation of standalone and non-standalone functions. Since all functions in HumanEval belong to standalone (*self-contained* and *slib-depend*), we further compare the effectiveness of three models in generating the standalone functions in CoderEval with the effectiveness of non-standalone functions. The *standalone* and *non-standalone* rows in Table 4 shows that in CoderEval for Python or CoderEval for Java, the effectiveness of the three models in generating the standalone functions is substantially higher than that of non-standalone functions. Considering that more than 70% of real functions in open-source projects belong to non-standalone functions, improving the model's ability



(a) CoderEval for Python (b) CoderEval for Java

Figure 3: Number of correctly-generated functions of CodeGen, PanGu-Coder, and ChatGPT on CoderEval

to consider and use contextual information correctly is vital for the practical value of such technology.

To reflect the three models' effectiveness at fine-grain dependency levels of CoderEval, we evaluate both standalone and non-standalone functions at fine-grain levels. The type of function represents the non-overlapping part between two neighboring runnable levels in Fig 2. For example, the *class-depend* represents those functions that 'must' depend on other code in the current class. Standalone functions include two dependency levels, *self-contained* and *slib-depend*, and non-standalone functions include four dependency levels, *plib-depend*, *class-depend*, *file-depend*, and *project-depend*. There are only five functions belonging to *plib-depend* and one function belonging to *file-depend* in CoderEval for Java. In CoderEval for Java, most functions depend on the contextual information within their own classes. We find that there are only five functions that not only depend on the contextual information in third-party libraries, but also do not depend on any contextual information within their own projects. On the other hand, in CoderEval for Java, there is only one function that depends on the contextual information within its file but outside its class. The reason is that Java mainly uses classes as its basic units. Java functions rely on only other static classes in the current file when these functions are *file-depend*. The preceding situation accounts for a relatively low proportion in open-source code for Java, so CoderEval collects only one method that relies on other static classes in the current file. From the table, we find that besides the Pass@5 and Pass@10 of *class-depend* on CoderEval for Java, ChatGPT almost outperforms the other two models on all depend levels. As previously analyzed in Section 5.1.1, the main reason is ChatGPT's model size is much larger than the other two models.

In summary, on both CoderEval for Python and CoderEval for Java, the effectiveness of the three models in generating standalone functions is substantially higher than that in generating non-standalone functions. Different models have their unique capabilities in generating code, and how to combine the code generation capabilities of different models is a worthwhile research direction.

Table 4: Effectiveness comparison between the generation of standalone and non-standalone functions on CoderEval

Depend-Level	Model	Python			Java		
		Pass@1	Pass@5	Pass@10	Pass@1	Pass@5	Pass@10
standalone:	CodeGen	18.1%	33.38%	38.1%	26.25%	46.4%	52.5%
	PanGu-Coder	19.52%	31.30%	38.10%	43.88%	57.23%	62.5%
	ChatGPT	35.87%	43.56%	47.62%	64.88%	68.65%	70.0%
- self-contained	CodeGen	22.57%	36.75%	40.00%	27.27%	49.37%	56.36%
	PanGu-Coder	23.71%	34.93%	40.00%	48.36%	57.94%	61.82%
	ChatGPT	52.29%	60.08%	82.86%	61.82%	66.21%	67.27%
- slib-depend	CodeGen	12.5%	29.17%	35.71%	24.0%	39.87%	44.0%
	PanGu-Coder	14.29%	26.76%	35.71%	34.0%	55.68%	64.0%
	ChatGPT	15.36%	22.92%	28.57%	71.6%	74.0%	76.0%
non-standalone:	CodeGen	6.23%	14.38%	17.96%	7.33%	17.17%	23.33%
	PanGu-Coder	8.92%	17.02%	23.35%	15.6%	26.81%	32.67%
	ChatGPT	15.45%	21.18%	23.35%	19.67%	28.97%	33.33%
- plib-depend	CodeGen	4.76%	16.16%	23.81%	0%	0%	0%
	PanGu-Coder	13.33%	22.75%	28.57%	0%	0%	0%
	ChatGPT	21.43%	28.06%	28.57%	0%	0%	0%
- class-depend	CodeGen	5.82%	9.05%	10.91%	8.3%	19.21%	26.0%
	PanGu-Coder	7.82%	15.04%	21.82%	19.9%	32.59%	40.0%
	ChatGPT	8.73%	12.57%	14.55%	22.4%	31.49%	36.0%
- file-depend	CodeGen	7.79%	20.19%	25.00%	0%	0%	0%
	PanGu-Coder	9.41%	19.04%	26.47%	0%	0%	0%
	ChatGPT	21.03%	29.09%	32.35%	0%	0%	0%
- project-depend	CodeGen	3.91%	8.33%	8.7%	6.14%	14.89%	20.45%
	PanGu-Coder	6.09%	10.51%	13.04%	7.95%	17.33%	20.45%
	ChatGPT	9.57%	12.08%	13.04%	16.14%	27.2%	31.82%

5.2 RQ2: How do different models perform in correctly generating the contextual information?

To further study the *context-awareness* of large language models, we analyze the identifier tokens in generated code and focus on three types of tokens: TypeReference, APIInvocation, and VarReference (we exclude variables defined in the current function).

5.2.1 Definition of Acc@K:. We define the Acc@K metric to evaluate the ability of models to generate contextual information expected in the oracle implementation. If one of the generated K samples contains the correct token, it is considered to hit the correct contextual information. Acc@K computes the percentage of correctly predicted tokens in all of the expected tokens in the oracle implementation, so it is actually the estimated recall of expected contextual dependency. The precision is not computed, since redundant tokens sometimes have no impact on semantics and behavior.

5.2.2 Overall Acc@k of three models. Fig 4 shows the Acc@k of three models, the results shown in the figure are the average values obtained from 10 experiments. We find it is consistent with Pass@k (i.e., the generated functions can pass test cases), and the Acc@k of ChatGPT is also the best one in generating contextual dependency. To further explore the ability of different models in generating different types of context information, we divide the context token into three categories: TypeReference, APIInvocation, and VarReference, which usually form the most important part of the implementation at runnable-levels with external dependency. As shown in Fig 4, on CoderEval for Python, all models show poor effectiveness in generating the correct "VarReference" token, while the effectiveness in generating the "TypeReference" token is relatively well. However, when it comes to Java, all models have poor effectiveness in generating the correct "TypeReference" token, while the effectiveness in generating "APIInvocation" tokens is the best. The experimental results indicate that the ability of large models to generate different types of contextual tokens varies across different languages. Generating tokens of different types for different languages is an interesting future research direction for code generation based on large language models.

Table 5: The similarity between the original docstring and human-labeled docstring

Metrics / Language	BLEU-4	Jaccard Similarity
Python	54.7	55.2
Java	18.1	26.9

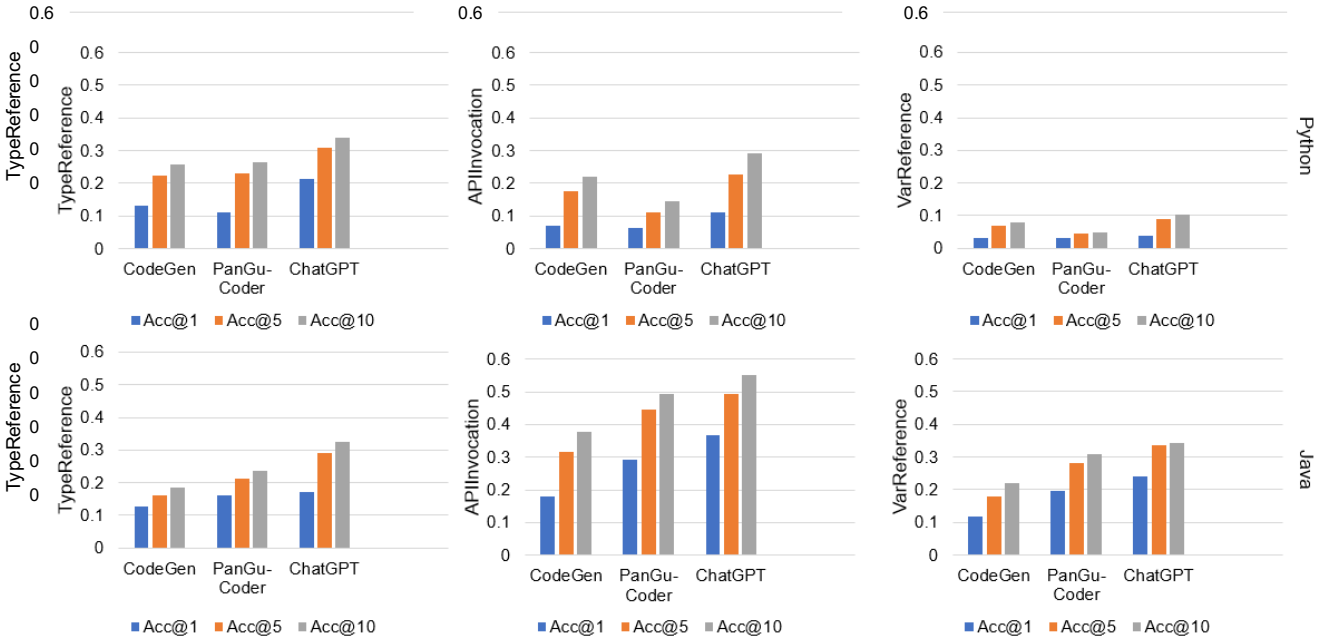


Figure 4: The accuracy of three models that they can correctly generate the contextual tokens

In summary, As listed in Table 3 and Fig 4, the ability of large language models to generate correct contextual information (Acc@k) is highly correlated with that for generating correct code (Pass@k). As shown in Fig 4, the ability of large language models to generate different types (i.e., TypeReference, APIInvocation, and VariableReference) of contextual tokens varies across different languages.

5.3 RQ3: How does the prompt affect the effectiveness of different models?

5.3.1 Motivation: There are three main reasons for including human-labeled docstrings in CoderEval. First, including human-labeled docstrings is to mitigate the memory effect (i.e., original docstrings have a high probability of being used by existing large language models in the pre-training stage) of language models and explore the impact of prompts with human labeling on the quality of code generated by large language models. Second, including human-labeled docstrings is to study how the model performs on different docstrings. We evaluate the text-level similarity between the built-in docstring of the function in CoderEval and the human-labeled docstring using Jaccard similarity and BLEU values. We find that the Jaccard similarity and BLEU values of the original docstring of the function and the human-labeled docstring are different in CoderEval for Python and CoderEval for Java. The original docstring of the function and the human-labeled docstring both describe the functionality of the code implementation. We want to explore through this experiment the ability of large language models to generate code for docstrings with different text levels but roughly the same semantics. Third, including human-labeled docstrings is to provide high-quality docstrings for the functions in CoderEval. In particular, 13 software engineers with at least 3-year experience of Python/Java additionally provide a human-labeled version of docstring for each problem in CoderEval. For the human-labeled docstring of each function, two developers independently write it, and ultimately the first author of this paper selects a high-quality docstring from them.

5.3.2 Overall effectiveness. We evaluate the three models on both the original and the human-labeled prompt and show the result in Table 6. Table 5 shows the similarity between the original docstring and human-labeled docstring. From Table 5, we find that the BLEU value and Jaccard similarity of the original docstring and human-labeled docstring in the function on CoderEval for Python is more than that on CoderEval for Java. Table 6 shows that the value of Pass@k of the results for the original docstring and human-labeled docstring on CoderEval for Python is also more similar than that on CoderEval for Java. Note that, the effectiveness of all models with original docstring and human-labeled docstring on CoderEval for Python is different from that with original docstring and human-labeled docstring on CoderEval for Java. On CoderEval for Java, the effectiveness of using the original docstring for CodeGen and ChatGPT is better than using human-labeled docstring, while their effectiveness order in PanguCoder is just the opposite. There are two reasons. First, the similarity between the original docstring and human-labeled docstring for Python is more similar than the similarity between the original docstring and human-labeled docstring for Java. Second, comparing the generated code with two prompt versions, we find that it is related to the proportion of the target language corpus in the whole training data. For Python, the stable effectiveness can be attributed to dedicated training or

Table 6: Effectiveness with two Prompt Versions of CoderEval

Model	Prompt	Python			Java		
		Pass@1	Pass@5	Pass@10	Pass@1	Pass@5	Pass@10
CodeGen	Original	9.48%	19.58%	23.48%	13.87%	27.12%	33.04%
	Human Label	12.26%	22.49%	25.65%	10.65%	21.36%	26.52%
PanGu-Coder	Original	11.83%	20.93%	27.39%	25.43%	37.39%	43.04%
	Human Label	13.74%	21.14%	24.78%	26.70%	40.33%	46.09%
ChatGPT	Original	21.13%	27.31%	30.00%	35.39%	42.77%	46.09%
	Human Label	26.61%	31.31%	32.61%	26.96%	34.85%	37.39%

fine-tuning on the Python code corpus, while for Java, only PanGu-Coder undergoes pre-training on the Java code corpus. Therefore, while multilingual pre-training can transfer knowledge and bring better generalization across languages, monolingual pre-training or fine-tuning is still necessary for the stability of models.

> In summary, we find that the choice of using the human-labeled docstring vs. the original docstring has an impact on code generation effectiveness. In the task of code generation for a single language, when given natural language descriptions with the same semantics but different expressions, the model trained with the single language corpus performs better than the model trained with multiple languages.

6 RELATED WORK

HumanEval is a benchmark to assess code generation models on the functional correctness of programs synthesized from docstrings [7]. It consists of 164 hand-written programming problems and solutions in Python, each of which includes a function signature, docstring, body, and multiple unit tests.

Following HumanEval, AiXBench [11] is proposed to benchmark code generation models for Java, it contains 175 samples for automated evaluation and a dataset of 161 samples for manual evaluation. The authors of AiXBench present a new metric for automatically assessing the correctness of code, and a set of criteria to manually evaluate the overall quality of the generated code.

MultiPL-E [5] is the first multi-language parallel benchmark for text-to-code-generation, it extends the HumanEval and MBPP [2] to support 18 programming languages. MultiPL-E is a suite of compilers and an evaluation framework for translating code generation benchmarks from Python into other programming languages. MultiPL-E translates unit tests, doctests, Python-specific terminology, and type annotations. MultiPL-E contains two parallel benchmarks for code generation in 18 languages encompassing various programming paradigms, language features, and popularity levels.

All preceding benchmarks contain only standalone functions. DS-1000 [16] is another benchmark test that contains 1000 questions, covering seven widely used Python data science libraries: NumPy, Pandas, TensorFlow, PyTorch, Scipy, Scikit-learn, and Matplotlib. The authors of the DS-1000 mitigate the problem of data leakage by manually modifying functions while emphasizing the use of data from real development scenarios to construct the DS-1000, which takes contextual information into account. DS-1000 contains 451 of the 1000 questions from the StackOverflow question.

Concode [14] is a new large dataset that contains over 100,000 examples of Java classes from open-source projects. Concode collects each function from a public Java project on GitHub that contains environment information as well as NL (Javadoc style method annotations) and code tuples. The authors of Concode collect Java files from approximately 33,000 repositories and then split them into training, validation, and test sets based on the repository, rather than purely random.

7 THREATS TO VALIDITY

One major threat in our work comes from the accuracy of the statistics for the proportion of standalone and non-standalone functions. Section 3.2.2 shows the approach that we used to count the proportion of standalone and non-standalone functions. To identify whether a function depends on a third-party library, we collect all third-party libraries from *pypi* for Python and *maven* for Java. Due to our inability to guarantee the collection of all third-party libraries, there may be a slight deviation in the statistical proportion of standalone and non-standalone functions.

8 CONCLUSION

In this paper, we have presented a new benchmark named CoderEval to evaluate a model's effectiveness in pragmatic code generation scenarios. Compared with the widely used HumanEval benchmark, CoderEval includes carefully selected programming tasks from various open-source projects to evaluate the effectiveness of models in pragmatic code generation. The experimental results show that CoderEval can reveal the strengths and weaknesses of the importance of three publicly available industrial models, highlighting the limitations of these existing code generation models in generating non-standalone functions.

ACKNOWLEDGMENTS

This work was partially supported by National Natural Science Foundation of China under Grant No. 62161146003, National Key Research and Development Program Project (2021YFF0704202), a grant from Huawei, and the Tencent Foundation/XPLORER PRIZE.

We would like to thank Jiaming Huang, Tianyu Chen, and Ziyue Hua for their help with improving the quality of unit test cases in CoderEval.

REFERENCES

[1] 2022. https://github.com/CoderEval/CoderEval.

[2] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. arXiv:cs.PL/2108.07732

[3] Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. 2021. *GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow.* https://doi.org/10.5281/zenodo.5297715

[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language Models Are Few-shot Learners. *Advances in Neural Information Processing Systems* (2020), 1877–1901.

[5] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. 2022. MultiPL-E: A Scalable and Extensible Approach to Benchmarking Neural Code Generation. *arXiv preprint arXiv:2208.08227* (2022).

[6] Shubham Chandel, Colin B Clement, Guillermo Serrato, and Neel Sundaresan. 2022. Training and Evaluating a Jupyter Notebook Data Science Assistant. *arXiv preprint arXiv:2201.12901* (2022).

[7] 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* (2021).

[8] 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. *arXiv preprint arXiv:2207.11280* (2022).

[9] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. InCoder: A Generative Model for Code Infilling and Synthesis. *arXiv preprint arXiv:2204.05999* (2022).

[10] Patrick Haluptzok, Matthew Bowers, and Adam Tauman Kalai. 2022. Language Models Can Teach Themselves to Program Better. *arXiv preprint arXiv:2207.14502* (2022).

[11] Yiyang Hao, Ge Li, Yongqiang Liu, Xiaowei Miao, He Zong, Siyuan Jiang, Yang Liu, and He Wei. 2022. AixBench: A Code Generation Benchmark Dataset. https://doi.org/10.48550/arXiv.2206.13179

[12] 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*, Vol. 1.

[13] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2019. The Curious Case of Neural Text Degeneration. *arXiv preprint arXiv:1904.09751* (2019).

[14] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping Language to Code in Programmatic Context. arXiv:cs.CL/1808.09588

[15] Joel Jang, Seonghyeon Ye, and Minjoon Seo. 2023. Can Large Language Models Truly Understand Prompts? A Case Study with Negated Prompts. In *Transfer Learning for Natural Language Processing Workshop*. 52–62.

[16] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2022. DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation. arXiv:cs.SE/2211.11501

[17] 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* (2023).

[18] Yujia Li, David 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. https://doi.org/10.48550/ARXIV.2203.07814

[19] 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* (2023).

[20] Naman Goyal Mike Lewis, Yinhan Liu. 2019. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. https://doi.org/10.48550/arXiv.1910.13461

[21] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A Conversational Paradigm for Program Synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[22] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving Language Understanding by Generative Pre-training. (2018).

[23] Advait Sarkar, Andrew D. Gordon, Carina Negreanu, Christian Pölitz, Sruti Srinivasa Ragavan, and Ben Zorn. 2022. What Is It Like to Program with Artificial Intelligence? (2022). https://doi.org/10.48550/arXiv.2208.06213

[24] John Schulman, Barret Zoph, Christina Kim, Jacob Hilton, Jacob Menick, Jiayi Weng, Juan Felipe Ceron Uribe, Liam Fedus, Luke Metz, Michael Pokorny, et al. 2022. ChatGPT: Optimizing Language Models for Dialogue. *OpenAI blog* (2022).

[25] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2022. Repository-level Prompt Generation for Large Language Models of Code. *arXiv preprint arXiv:2206.12839* (2022).

[26] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *Advances in Neural Information Processing Systems* 30 (2017).

[27] Thomas Wang, Adam Roberts, Daniel Hesslow, Teven Le Scao, Hyung Won Chung, Iz Beltagy, Julien Launay, and Colin Raffel. 2022. What Language Model Architecture and Pretraining Objective Work Best for Zero-Shot Generalization? *arXiv preprint arXiv:2204.05832* (2022).

[28] Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2022. Execution-Based Evaluation for Open-Domain Code Generation. *arXiv preprint arXiv:2212.10481* (2022).

[29] Wei Zeng, Xiaozhe Ren, Teng Su, Hui Wang, Yi Liao, Zhiwei Wang, Xin Jiang, ZhenZhang Yang, Kaisheng Wang, Xiaoda Zhang, Chen Li, Ziyan Gong, Yifan Yao, Xinjing Huang, Jun Wang, Jianfeng Yu, Qi Guo, Yue Yu, Yan Zhang, Jin Wang, Hengtao Tao, Dasen Yan, Zexuan Yi, Fang Peng, Fangqing Jiang, Han Zhang, Lingfeng Deng, Yehong Zhang, Zhe Lin, Chao Zhang, Shaojie Zhang, Mingyue Guo, Shanzhi Gu, Gaojun Fan, Yaowei Wang, Xuefeng Jin, Qun Liu, and Yonghong Tian. 2021. PanGu-α: Large-scale Autoregressive Pretrained Chinese Language Models with Auto-parallel Computation. *CoRR* abs/2104.12369 (2021). arXiv:2104.12369 https://arxiv.org/abs/2104.12369

[30] Kaiyang Zhou, Jingkang Yang, Chen Change Loy, and Ziwei Liu. 2022. Learning to Prompt for Vision-language Models. *International Journal of Computer Vision* (2022), 2337–2348.

[31] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2022. Large Language Lodels Are Human-level Prompt Engineers. *arXiv preprint arXiv:2211.01910* (2022).