# NaturalCodeBench: Examining Coding Performance Mismatch on HumanEval and Natural User Queries

**Shudan Zhang**[12†*], **Hanlin Zhao**[1*], **Xiao Liu**[12*], **Qinkai Zheng**[12*], **Zehan Qi**[12†], **Xiaotao Gu**[1], **Yuxiao Dong**[2], **Jie Tang**[2]

[1]Zhipu.AI [2]Tsinghua University

## Abstract

Large language models (LLMs) have manifested strong ability to generate codes for productive activities. However, current benchmarks for code synthesis, such as HumanEval, MBPP, and DS-1000, are predominantly oriented towards introductory tasks on algorithm and data science, insufficiently satisfying challenging requirements prevalent in real-world coding. To fill this gap, we propose NATURALCODEBENCH (NCB), a challenging code benchmark designed to mirror the complexity and variety of scenarios in real coding tasks. NCB comprises 402 high-quality problems in Python and Java, meticulously selected from natural user queries from online coding services, covering 6 different domains. Noting the extraordinary difficulty in creating testing cases for real-world queries, we also introduce a semi-automated pipeline to enhance the efficiency of test case construction. Comparing with manual solutions, it achieves an efficiency increase of more than 4 times. Our systematic experiments on 39 LLMs find that performance gaps on NCB between models with close HumanEval scores could still be significant, indicating a lack of focus on practical code synthesis scenarios or over-specified optimization on HumanEval. On the other hand, even the best-performing GPT-4 is still far from satisfying on NCB. The evaluation toolkit and development set are available at `https://github.com/THUDM/NaturalCodeBench`.

## 1 Introduction

Large language models (LLMs) pre-trained on extensive open code repositories (Chen et al., 2021; OpenAI et al., 2023; Li et al., 2023a; Chowdhery et al., 2023) have demonstrated impressive performance on code synthesis and even achieve performance comparable to average human level in coding competitions (Li et al., 2022). Unlike open text generation, which often underscores human preferences as noted by (Ouyang et al., 2022), code synthesis prioritizes accuracy and the fulfillment of user intent, essential for practical production and application.

As a result, evaluating code synthesis presents unique challenges in the era of LLMs. Traditional evaluation metrics by token matching (Papineni et al., 2002; Lin, 2004; Popović, 2015) show a weak correlation with human judgement (Evtikhiev et al., 2023) and overlook functional correctness of the generated code (Eghbali and Pradel, 2023; Tran et al., 2019). Recently, execution-based evaluation has gained increasing popularity, where code generated by models is tested through unit tests to verify its functional correctness. It leads to the development of several benchmarks, including HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), MBXP (Athiwaratkun et al., 2023), CodeContests (Li et al., 2022), and DS-1000 (Lai et al., 2023).

Notwithstanding their commendable reliability and accuracy, these benchmarks fall short to sufficiently capture the wide range of needs and complexity found in real-world engineering applications. They are primarily limited to well-defined coding problems in algorithm, program basics, or data science. For example, as shown in Figure 1, a problem from HumanEval (Chen et al., 2021) tests the implementation of a basic function `has_close_elements` and takes floating-point arguments as inputs. However, in practical applications, user engineering requirements can be much more complex and varied. In Figure 1, we showcase an example adapted from a real user query, where the user asks to read and parse XML files given certain tags. Difficult and costly though it is, curating a benchmark composed of such problems is meaningful for evaluating the real user experience of LLM code synthesis.

**Contributions.** In light of the challenge, we intro-

*SZ, HZ, XL, and QZ contributed equally.
†Work done when SZ and ZQ interned at Zhipu AI.



*Findings of the Association for Computational Linguistics ACL 2024*, pages 7907–7928
August 11-16, 2024 

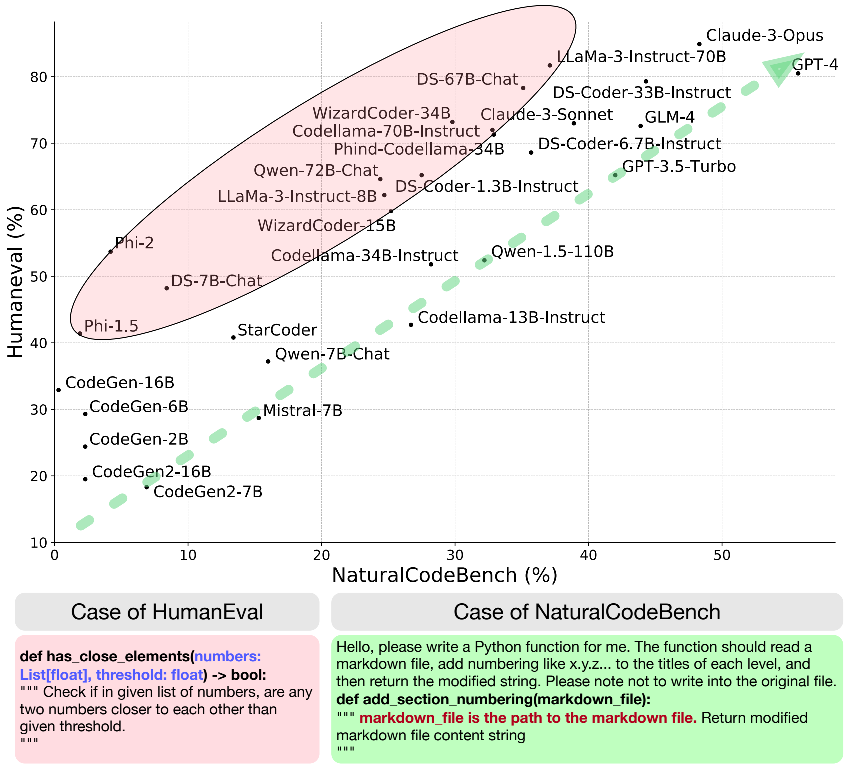


Figure 1: Comparing HumanEval and NaturalCodeBench. In the scatter plot, the x-axis represents the NaturalCodeBench score, and the y-axis indicates the HumanEval performance of various LLMs.

duce NaturalCodeBench (NCB), a challenging application-driven dataset for code synthesis evaluation. NCB is dedicated to creating a reliable evaluation environment that is more aligned with real-world applications. We leverage an CodeGeeX (Zheng et al., 2023b) online services to collect real and diverse application-related user queries. After filtering and reprocessing, 402 high-quality Python and Java problems are compiled, covering 6 domains including software, front-end, system administration, and artificial intelligence, highlighting practical scenarios. Beyond basic data structures like lists and numbers, the test inputs for NCB problems include versatile file types and other complex structures, making it more challenging.

The challenging nature of NCB necessitates significant human labor in its annotation process To improve construction efficiency, we tailor a semi-automated annotation pipeline to curate high-quality, testable, and useful queries with corresponding test cases. Specifically, we employ GPT-4 (OpenAI et al., 2023) to generate reference solutions followed by manual correction. Subsequently, GPT-4, guided by the problem descriptions and reference solutions, generates multiple test cases, which are also refined with manual correction, for each problem. Consequently, the annotators are only required to correct any errors, substantially reducing the time and manpower required. Comparative experiments reveal that our semi-automated pipeline can quadruple the construction speed of the evaluation framework, as evidenced by tests involving programming experts with or without the pipeline.

Based on NCB, we conduct extensive experiments on a variety range of LLMs, encompassing 39 APIs or open models. The results indicate that although certain LLMs demonstrate comparable performance on established benchmarks like HumanEval, they exhibit significant performance disparities when evaluated using NCB. It suggests that there may be inadequate focus on optimizing LLMs for practical coding applications, or have conducted over-specified optimization on HumanEval-style problems. More importantly, even the best-performing GPT-4 only reaches about a pass rate of 53%, demonstrating a large room for LLMs to improve their coding skills to face real-world coding challenges.

To facilitate community research, we pack up the whole NCB testing environment into a docker image and make its development set publicly available. To sum up our contributions:

- We propose NATURALCODEBENCH, a benchmark that aligns with real-world applications, comprising 402 problems in Python and Java across 6 domains. We open source 140 problems (70 Python, 70 Java) as the development set of NCB for research purposes, but keep the 262 problems of the test set closed to avoid contamination.
- We introduce a semi-automated pipeline for the construction of code synthesis benchmarks, which significantly reduces time and manpower costs without compromising the quality of test cases. Comparative experiments reveal that our semi-automated pipeline can quadruple the construction speed of the evaluation framework
- We systematically benchmark the code generation capabilities of 39 LLMs using NCB. Besides quantitative evaluation, we carry out a deep insight into the present stage of development in LLMs for code generation, and outline potential pathways for future progress.

## 2 Benchmark Construction

The overview of NCB is shown in Figure 2. The pipeline of constructing NCB consists of four steps: 1) collecting and filtering high-quality problems from online services (Section 2.1) 2) constructing a complete evaluation framework through a semi-automated pipeline (Section 2.2) 3) designing prompts to align different models (Section 2.3) 4) translating all problems and instructions to produce bilingual versions (Section 2.4).

### 2.1 Problem Selection

**Collecting Real-World Problems.** To establish a meaningful and practical benchmark, we centered on collecting real-world code problems frequently encountered by users. To achieve this, the seed problems of NCB are cleaned from the queries in coding online services. A part of users have granted permission for their data to be utilized exclusively for research purposes. We have strictly adhered to this directive by collecting only the relevant data from these consenting users and have implemented robust de-identification measures to eliminate any possibility of information leakage. We collect a varied collection of queries, spanning multiple programming languages, problem types, and levels of complexity. This diversity ensures that our benchmark accurately reflects a broad range of code issues users encountering in practice. We specifically concentrated on queries related to Python and Java, chosen for their widespread use in different domains.

**Filtering Testable Problems.** While it's possible to source inexhaustible queries from online services, many of these queries posed by users are either of low value or challenging to test the solution of these queries. For instance, some users may only seek basic clarifications on a built-in function, while others may not clearly articulate their objectives. To sieve out unsuitable queries for our testing, we've implemented a two-step filtering process. Initially, we employ GPT-3.5 to filter out low-quality queries, which saves on labour. This is achieved by adding specific criteria in the instruction, instructing GPT-3.5 to abandon those problems that cannot meet all specified requirements. These criteria are as follows: 1) Each query must involve at least one task, where the user requests the model's assistance in solving one or more problems. 2) Each query should be associated with several input-output pairs, ensuring that a given input correspond to a singular, definitive output. 3) The query must not contain any elements of randomness or uncertainty. The specifics of the instruction are detailed in (Appendix A). Following this automated prescreening, we conduct a manual review to further refine the selection, adhering to the outlined criteria. This process yields a final set of 201 unique Python and 201 unique Java problems. It is noteworthy that over 80% of the initial queries failed to meet our stringent requirements.

### 2.2 Semi-automated Pipeline

In this section, we will introduce our semi-automated pipeline. To generate structurally complex and accurate test cases by GPT-4, it is first necessary to determine the arguments and return values of functions, as well as the names of objects. Therefore, a completely accurate reference solution is required initially. We generate a solution using GPT-4, then manually correct all errors. After this, based on the problem description and the reference solution, we instruct GPT-4 to generate multiple test cases. These are then reviewed by programming experts who correct errors and supplement any deficiencies in the generated test cases.

**Generating and Rewriting Reference Solution.**

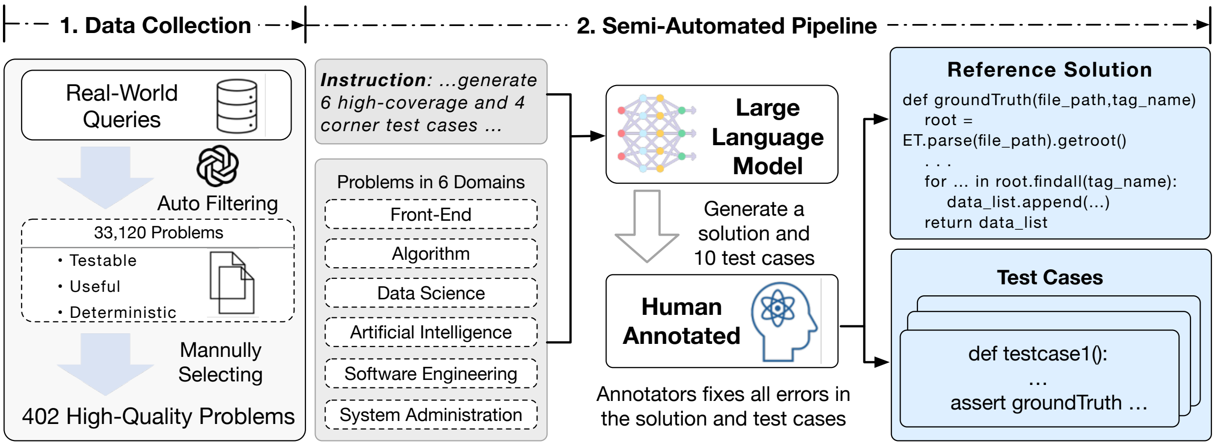


Figure 2: Overview of NaturalCodeBench. 1) Data Collection: collecting real-world queries from coding online services and selecting high-quality problems from the queries by GPT-3.5 and human annotators. 2) Semi-Automated Pipeline: improving efficiency of constructing evaluation framework by generating a solution and test cases with LLMs and then having them corrected by human annotators.

GPT-4 is instructed to generate a solution for each problem in NCB. It is important to note that while GPT-4 is highly capable, it is not infallible. Therefore, each solution generated by GPT-4 is meticulously examined by expert programmers to ensure correctness. In cases where the generated code contains errors, the expert programmers rewrite the code to rectify these issues. This process ensures the quality of the reference solutions. Even though we did not use the reference solution in NCB for evaluation, we provided them to facilitate the generation of test cases and future research.

**Build High-Coverage and Corner Evaluation.** We employ GPT-4 to generate evaluation codes for each problem. We construct a prompt using 1) the description of the problem for GPT-4 to inspect; 2) the reference solution to demonstrate the names and formats in the code; 3) an instruction to encourage GPT-4 to come up with effective test cases. Specifically, each prompt start with an instruction that ask GPT-4 to produce ten test cases based on the description of problem and the reference solution. Then, we present both the description of problem and its reference solution. We finalize the prompt with an initial segment of the evaluation code to assist GPT-4 in accurately generating the desired code format. Our objective is to harness GPT-4's advanced comprehension and analytical abilities to learn valid format in the code and essential functionalities of the reference solution to enable the generation of superior test cases that are adept at uncovering latent errors in code.

A complete and effective test should seek to identify potential bugs at different locations in the code, while also finding inputs that might trigger errors in the code. High coverage ensures that each test case examines more code and branches, thereby facilitating the discovery of concealed errors. Meanwhile, it is often observed that corner values in a problem's input are most prone to trigger code errors. Consequently, our instruction will cause some of the test cases generated by GPT-4 to have higher coverage, while the other part will be some corner values contained in the problem, so as to obtain more effective test cases.

Subsequently, expert programmers review and correct any test cases with formatting and answer errors. To ensure that the final evaluation framework is error-free.

## 2.3 Alignment Between Different Models

In contrast to the problem format in HumanEval, the majority of problems in our benchmark are composed in natural language by actual users. Consequently, there is no predetermined naming convention for functions or classes created by models. This divergence can lead to inconsistencies between the names generated by LLMs and those referenced in test cases. To address this issue of name misalignment, we present a representative test case that includes the designated function or class name and its usage within the test. We then instruct the LLMs to adhere to the naming convention specified in the provided test case when generating solutions. It is important to note that the test cases utilized for solution generation are not employed in subsequent testing phases. The details of the instruction is showed in Appendix A.

### 2.4 Building Bilingual Benchmark

The majority of the questions we collected from online services are in Chinese, which is not fair for the LLMs that are primarily designed for English. Therefore, we translate all the problems, resulting in both Chinese and English versions.

## 3 Dataset Statistics

We provide more detailed statistics in Table 2. NCB comprises a total of 402 problems collected from online services, with 201 problems in Python and 201 in Java, spanning across 6 domains: Database, Artificial Intelligence, Data Science, Algorithm and Data Structure, Front-End, Software Engineering, and System Administration. This diversity also leads to complex input data types in NCB, which are classified into 9 categories: number (int/float/boolean), string, list (array), dict, tensor (matrix), data frame (table), plain text file, image, and special format file. The first four are the most common and simplest data types. Since a boolean can be represented by 1 and 0, we consider it as a type of number. Matrix and list are two similar types of data, but they are categorized separately due to differences in their usage scenarios. Due to the current popularity of deep learning, tensor has become a very common data format. Therefore, we have designated a separate category for tensor and have included matrix within this category. The last three are all file types, differentiated by their processing methods. The content of a plain text file is text and can be directly read. Figures require processing of each pixel value. A special format file refers to files that require specific methods for processing, such as PDF and DOCX.

Each problem within the dataset has been carefully curated with a set of test cases to assess the correctness of solutions. On average, there are 9.3 test cases associated with each problem. These cases are strategically designed, with about 60% focused on enhancing statement and branch coverage, and the remaining 40% dedicated to evaluating the robustness of solutions against corner values. The average word count for each problem in the NCB is 78.3.

**Compared with Other Benchmark.** Table 1 compares NCB to other benchmarks. It is noteworthy that our benchmark offers a substantial supplement to current benchmarks in terms of both problem and data types. Unlike HumanEval and MBPP, which consist of 96.9% and 89.5% algorithmic and basic programming problems respectively, our benchmark features a more balanced distribution across each domain.

In addition, NCB includes more data types. Furthermore, NCB focuses on assessing the model's ability to handle multiple file formats, a type of data that is both very commonly used in daily life and relatively challenging to process. We note that the problems involving files have fewer test cases, since GPT-4 still struggles to fully generate various types of file . This is also more challenging for human annotators to design compared to simpler data types.

On the other hand, NCB is also limited by its size due to the high costs of problems collection and the construction of the evaluation framework. We are continuously working on expanding our benchmark.

## 4 Experiments

### 4.1 Setup

We conducted comprehensive evaluations of 39 popular state-of-the-art models. For proprietary models, our focus was on OpenAI's GPT-4-Turbo-0125, GPT-4-Turbo-1106, GPT-4, GPT-3.5-Turbo, Anthropic's Claude-2, ZhipuAI's CodeGeeX3. In the case of open-source models, we performed evaluations using the vLLM (Kwon et al., 2023) and FastChat (Zheng et al., 2023a) framework. Our evaluation primarily utilizes pass@k (Chen et al., 2021) as the metric to accurately assess the functional correctness of code generated by these models. For k equal to 1, we employ greedy-search decoding. For random sampling, we demonstrate the best pass@k results of the best-performing models with each LLM family for each $k \in \{10, 50\}$, where the sampling temperature is set to 0.2 and topp to 0.9.

Our semi-automated pipeline is capable of reducing the time required for benchmark construction without compromising the quality of test cases. This paper primarily focuses on evaluating the efficiency of benchmark construction and the quality of test cases. Specifically, we adopt code coverage (Ivanković et al., 2019), a widely used metric for assessing the effectiveness of testing, as the criterion for evaluating the quality of test cases. We invite five programming experts, each tasked with constructing the same five problems. Initially, we

| Benchmark | Instruction Information | | | | | Evaluation | |
|---|---|---|---|---|---|---|---|
| | #Problem | Domain | #Data Type | #Word | Source | #Test Case | Method |
| Humaneval (Chen et al., 2021) | 164 | Algorithm | 5 | 23.0 | Hand-Written | 7.7 | Test-Case |
| MBPP (Austin et al., 2021) | 974 | Program Basics | 5 | 15.7 | Hand-Written | 3.0 | Test-Case |
| DS-1000 (Lai et al., 2023) | 1,000 | Data Sci. | 6 | 140.0 | StackOverflow | 1.6 | Test-Case + SFC. |
| APPS (Hendrycks et al., 2021a) | 10,000 | Algorithm | 5 | 293.2 | Competitions | 13.2 | Test-Case |
| Humaneval+ (Liu et al., 2023a) | 164 | Algorithm | 5 | 23.0 | Hand-Written | 764.1 | Augmented Test Cases |
| **NaturalCodeBench** | **402** | **Application** | **6** | **78.3** | **Online Services** | **9.3** | **Test-Case** |

Table 1: Comparison between NATURALCODEBENCH and other benchmarks for code generation.

| | #Problems | | | Avg. #Test Cases | | |
|---|---|---|---|---|---|---|
| Dataset | Test | Dev | Total | Test | Dev | Total |
| Software | 88 | 44 | 132 | 9.7 | 8.6 | 9.3 |
| Data Sci. | 68 | 32 | 100 | 9.6 | 8.6 | 9.3 |
| Algorithm | 73 | 22 | 95 | 9.5 | 8.8 | 9.3 |
| Sys. Admin. | 17 | 16 | 33 | 9.6 | 8.5 | 9.1 |
| AI. System | 13 | 15 | 28 | 9.6 | 9.1 | 9.3 |
| Front-End | 3 | 11 | 14 | 10.0 | 8.7 | 9.0 |
| **Total/Avg.** | **262** | **140** | **402** | **9.6** | **8.7** | **9.3** |

Table 2: Detailed statistics of NATURALCODEBENCH.

ask each expert to manually write a standard solution and 5 test cases. Subsequently, for the same problems, they complete the writing of standard solutions and test cases using the semi-automated pipeline. As it is challenging to ensure identical test case coverage, we require that the test cases written under both methods should not have a code coverage of less than 80%. Then, for the sake of convenient comparison, we calculate the scores for each construction method in a straightforward manner, which is outlined as follows:

$$Score = \frac{LineCov. + BranchCov.}{TimeCost} * 10$$

## 4.2 Results of LLMs

Table 3 and Table 6 shows the pass@1 results on the test set and dev set of NCB, respectively. Considering the high consistency of results, we primarily analyze the results on the test set. As expected, OpenAI's GPT-4 achieves the highest score of 52.8%. The performance of GPT-4-Turbo is very close to that of GPT-4, differing only by 1.3% , with GPT-4-Turbo performing better in Java but showing a larger difference in Python. Among the open-source models, DeepSeek-Coder-33B-Instruct performs the best, reaching a score of 43.0%. However, the 9.8% score gap with GPT-4 remains significant. On the other hand, it surpasses the 40.7% achieved by GPT-3.5, exceeding it by 2.3%. In summary, the performance of state-of-the-art open-source models is now between GPT-3.5 and GPT-4, yet the majority of open-source models still do not match the performance of GPT-3.5.

When compared to a perfect score of 100%, it is observed that even the best-performing model, GPT-4, still falls significantly short. This is in contrast to its performance in HumanEval, where it has approached 90%.

Comparing the performance of models in Chinese and English versions, it is evident that the vast majority of models perform better in English. This holds true even for the top models, GPT-4 and GPT-4-Turbo, which outperform their average scores in Chinese by 1.1% and 3.9%, respectively.

Furthermore, Table 3 systematically presents the performance of various open-source models at different scales. Models smaller than 10B scored between 0.0% and 23.9%, models between 10B and 30B scored between 3.9% and 35.1%, models between 30B and 60B scored between 21.8% and 43.0%, and models larger than 60B scored between 27.9% and 33.2%. It is evident that the scale of the model still has a significant impact on performance. Larger models generally outperform smaller models, indicating that increasing scale can indeed enhance a model's capabilities. However, this is not to say that scale is everything; more refined data and training strategies can also significantly impact a model's performance. Some smaller models, such as DeepSeek-Coder-6.7B-Instruct, can outperform those larger than 30B by approximately 2.8% and those larger than 60B by approximately 1.9%.

Table 5 shows the pass@k results of best-performing LLMs with each LLM family on NCB, where $k \in \{10, 50\}$. We found that under random sampling, the scores of some models increased significantly. For instance, Codellama-70B-Instruct, unlike its performance on pass@1, clearly outperformed GPT-3.5 on both Pass@10 and Pass@50.

| Model | Size | NCB (zh) Python | Java | Total | NCB (en) Python | Java | Total | NCB Total Score | Rank | HumanEval Score | Rank | ΔRank |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| API LLMs | | | | | | | | | | | | |
| **GPT-4** (OpenAI et al., 2023) | N/A | 53.4 | 51.1 | 52.3 | 55.7 | 51.1 | 53.4 | 52.8 | 1 | 80.5 | 5 | 4 |
| **GPT-4-Turbo-0125** (OpenAI et al., 2023) | N/A | 51.4 | 58.6 | 55.0 | 48.6 | 51.4 | 50.0 | 52.5 | 2 | 87.2 | 1 | -1 |
| **GPT-4-Turbo-1106** (OpenAI et al., 2023) | N/A | 47.3 | 51.9 | 49.6 | 51.9 | 55.0 | 53.5 | 51.5 | 3 | 81.7 | 3 | 0 |
| **GPT-3.5-Turbo** (OpenAI, 2022) | N/A | 39.7 | 38.9 | 39.3 | 42.0 | 42.0 | 42.0 | 40.7 | 8 | 65.2 | 18 | 10 |
| **Claude-3-Opus** (Anthropic, 2023b) | N/A | 45.0 | 50.4 | 47.7 | 48.9 | 48.9 | 48.9 | 48.3 | 4 | 84.9 | 2 | -2 |
| **Claude-3-Sonnet** (Anthropic, 2023b) | N/A | 44.6 | 35.5 | 40.1 | 40.5 | 35.1 | 37.8 | 38.9 | 9 | 73.0 | 11 | 2 |
| **Claude-3-Haiku** (Anthropic, 2023b) | N/A | 41.3 | 35.9 | 38.6 | 36.9 | 30.5 | 33.7 | 36.2 | 11 | 75.9 | 9 | -2 |
| **Claude-2.1** (Anthropic, 2023a) | N/A | 33.6 | 32.8 | 33.2 | 34.4 | 36.6 | 35.5 | 34.4 | 13 | 71.2 | 16 | 3 |
| **GLM-4** (Zeng et al., 2023; Du et al., 2022) | N/A | 43.5 | 45.3 | 44.4 | 41.5 | 45.3 | 43.4 | 43.9 | 5 | 72.6 | 12 | 7 |
| **Gemini-1.5-Pro** (Blog, 2024) | N/A | 41.5 | 43.1 | 42.3 | 45.0 | 39.7 | 42.3 | 42.3 | 7 | 71.9 | 14 | 7 |
| **CodeGeeX3** (Zheng et al., 2023b) | N/A | 29.0 | 29.0 | 29.0 | 36.6 | 32.8 | 34.7 | 31.9 | 18 | 69.5 | 17 | -1 |
| Open LLMs | | | | | | | | | | | | |
| **Deepseek-Coder-Instruct** (Guo et al., 2024) | 33B | 44.3 | 38.9 | 41.6 | 44.3 | 44.3 | 44.3 | 43.0 | 6 | 79.3 | 6 | 0 |
| | 6.7B | 38.9 | 29.8 | 34.4 | 35.9 | 35.9 | 35.9 | 35.1 | 12 | 78.6 | 7 | -5 |
| | 1.3B | 18.3 | 24.4 | 21.4 | 27.5 | 25.2 | 26.4 | 23.9 | 22 | 65.2 | 19 | -3 |
| **Llama-3-Instruct** (AI@Meta, 2024) | 70B | 39.1 | 34.4 | 36.7 | 35.4 | 39.7 | 37.5 | 37.1 | 10 | 81.7 | 4 | -6 |
| | 8B | 35.9 | 21.5 | 28.7 | 19.7 | 21.7 | 20.7 | 24.7 | 21 | 62.2 | 21 | 0 |
| **Deepseek-Chat** (DeepSeek-AI, 2024) | 67B | 35.9 | 28.2 | 32.1 | 35.1 | 33.6 | 34.4 | 33.2 | 14 | 78.3 | 8 | -6 |
| | 7B | 3.8 | 12.2 | 8.0 | 8.4 | 19.1 | 13.8 | 10.9 | 30 | 48.2 | 26 | -4 |
| **Codellama-Instruct** (Roziere et al., 2023) | 70B | 35.1 | 32.1 | 33.6 | 32.8 | 30.5 | 31.7 | 32.6 | 15 | 72.0 | 13 | -2 |
| | 34B | 23.7 | 17.6 | 20.7 | 28.2 | 17.6 | 22.9 | 21.8 | 24 | 51.8 | 25 | 1 |
| | 13B | 20.6 | 16.8 | 18.7 | 26.7 | 19.1 | 22.9 | 20.8 | 25 | 42.7 | 26 | 1 |
| | 7B | 16.8 | 17.6 | 17.2 | 21.4 | 17.6 | 19.5 | 18.4 | 26 | 34.8 | 31 | 5 |
| **Phind-Codellama** (Phind, 2023) | 34B | 34.4 | 29.0 | 31.7 | 33.6 | 32.1 | 32.9 | 32.3 | 16 | 71.3 | 15 | -1 |
| **Qwen-1.5** (Bai et al., 2023a) | 110B | 35.4 | 28.2 | 31.8 | 38.5 | 26.7 | 32.6 | 32.2 | 17 | 52.4 | 24 | 7 |
| **Qwen-Chat** (Bai et al., 2023b) | 72B | 28.2 | 29.8 | 29.0 | 24.4 | 29.0 | 26.7 | 27.9 | 19 | 64.6 | 20 | 1 |
| | 7B | 11.5 | 13.0 | 12.3 | 16.0 | 11.5 | 13.8 | 13.0 | 28 | 37.2 | 30 | 2 |
| **WizardCoder** (Luo et al., 2023) | 34B | 24.4 | 22.9 | 23.7 | 29.8 | 22.1 | 26.0 | 24.8 | 20 | 73.2 | 10 | -10 |
| | 15B | 29.0 | 17.6 | 23.3 | 25.2 | 19.1 | 22.2 | 22.7 | 23 | 59.8 | 22 | -1 |
| **StarCoder** (Li et al., 2023a) | 15.5B | 13.0 | 13.0 | 13.0 | 16.8 | 9.9 | 13.4 | 13.2 | 27 | 40.8 | 29 | 2 |
| **Mistral-Instruct** (Jiang et al., 2023a) | 7B | 7.6 | 9.9 | 8.8 | 11.5 | 19.1 | 15.3 | 12.0 | 29 | 28.7 | 34 | 5 |
| **CodeGen2** (Nijkamp et al., 2023a) | 16B | 0.8 | 11.5 | 6.2 | 2.3 | 13.0 | 7.7 | 6.9 | 31 | 19.5 | 36 | 5 |
| | 7B | 2.3 | 5.3 | 3.8 | 6.9 | 5.3 | 6.1 | 5.0 | 32 | 18.3 | 37 | 5 |
| | 3.7B | 0.0 | 0.0 | 0.0 | 0.0 | 3.1 | 1.6 | 0.8 | 38 | 15.9 | 38 | 0 |
| | 1B | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 39 | 11.0 | 39 | 0 |
| **Phi** (Li et al., 2023b) | 2.7B | 5.3 | 3.1 | 4.2 | 3.1 | 5.3 | 4.2 | 4.2 | 33 | 53.7 | 23 | -10 |
| | 1.3B | 0.0 | 0.8 | 0.4 | 3.8 | 0.0 | 1.9 | 1.2 | 37 | 41.4 | 28 | -9 |
| **CodeGen** (Nijkamp et al., 2023b) | 16B | 0.8 | 5.3 | 3.1 | 0.3 | 9.2 | 4.8 | 3.9 | 34 | 32.9 | 32 | -2 |
| | 6B | 0.0 | 0.0 | 0.0 | 2.3 | 3.8 | 3.1 | 1.5 | 35 | 29.3 | 33 | -2 |
| | 2B | 0.0 | 0.0 | 0.0 | 2.3 | 3.8 | 3.1 | 1.5 | 36 | 24.4 | 35 | -1 |

Table 3: Evaluating LLMs on the test set of NATURALCODEBENCH. All results are pass@1 on greedy decoding. Dev set results are reported in Table 6. Compared to HumanEval (Chen et al., 2021), some LLMs present significant variations

We compared the Python scores on the test set of NCB with the performances of models on HumanEval, as shown in the Figure 1. Most models are located in the upper triangular area of the graph, with many models scoring high on HumanEval but exhibiting relatively lower performance on NCB.

### 4.3 Performance mismatch on HumanEval and NCB

We show the rank orders of all tested LLMs in Table 3 with regard to HumanEval and NCB, as well as the difference of rank orders. We also plot the corresponding performances on two benchmarks to scatter diagram in Figure 1. Based on the table and figure, we have some interesting findings.

Performances of most LLMs on two benchmarks grow linearly proportional, and the differences of scores' rank order are around 0. It demonstrates that NCB can indeed reflect the coding abilities of LLMs as HumanEval does in most cases.

However, we observe that some model series, notably the Phi, Deepseek-Chat, and WizardCoder, consistently exhibit a propensity to achieve superior rankings on the HumanEval dataset as opposed to the NCB across various scales, as shown in the Table 3. Additional model families, including CodeGen and Llama-3-Instruct, similarly display the trend, though to a reduced degree.

There might be a few potential hypotheses for the observation. First, as problems in NCB are more difficult and derived from natural user prompts, compared to those in HumanEval, LLMs with poorer generalization and instruction-following capabilities tend to perform worse. We find in preliminary experiments that problems in NCB cannot be properly solved by pre-trained base LLMs via mere in-context learning as HumanEval does, which indicates that to solve NCB problems requires stronger alignment and generalizability than HumanEval needs.

Second, it is possible that training sets of some LLMs are over-specifiedly optimized for HumanEval-style problems. On one hand, pre-training data of certain LLMs may be contaminated. As GPT-4 (OpenAI et al., 2023) reported, 25% of HumanEval has been contaminated in their pre-training corpus. On the other hand, instruction fine-tuning dataset may also be polluted. For example, Phi (Li et al., 2023b) reports a considerable amount of synthetic prompts resonating to some test samples in HumanEval. In (Yang et al., 2023b), the authors report leakage unidentifiable by n-gram overlap when using popular rephrasing techniques to create training sets. The performance discrepancy between HumanEval and NCB in our experiments is also an evidence of the potential contamination.

### 4.4 Results of Semi-automated Construction

In Table 4, we can observe that the coverage of hand-written test cases is almost identical to that of test cases constructed through a semi-automatic pipeline, yet the time required for the former significantly exceeds the time needed for constructing test cases via the semi-automatic pipeline. Specifically, test cases can be constructed via the semi-automated pipeline in just 40 minutes, whereas manual writing requires 175.9 minutes, a difference of more than 4x. Consequently, the scores obtained for test cases constructed using the semi-automated pipeline are far higher than those for manually written test cases, with an average difference of 37.6. In summary, constructing test cases through the semi-automatic framework can achieve significantly higher efficiency without substantial loss in quality compared to manual writing.

## 5 Related Work

**LLMs for code.** Significant advancements in LLMs (Vaswani et al., 2017, Devlin et al., 2019, Brown et al., 2020) are transforming everyday life, particularly in the field of coding, driven by the vast amount of openly available codebases and the push to enhance productivity among developers. Code-specific LLMs have proven their ability to perform various tasks such as code generation (Chen et al., 2021, Iyer et al., 2018, Li et al., 2022), program repair (Jiang et al., 2023b, Wei et al., 2023, Xia et al., 2023, Xia and Zhang, 2022), automated testing (Deng et al., 2023a, Deng et al., 2023b, Liu et al., 2023c, Xia et al., 2024, Yang et al., 2023a), code translation (Roziere et al., 2020, Roziere et al., 2022) and code summarization (Ahmed and Devanbu, 2023, Lu et al., 2021). Notably, prominent LLMs including CODEX (Chen et al., 2021), CodeGen (Nijkamp et al., 2023b), INCODER (Fried et al., 2023), and PolyCoder (Xu et al., 2022) have been developed and rigorously tested, particularly in code generation. This area, often referred to as the ultimate goal in computer science research since the early days of AI in the 1950s, involves the model producing code snippets from natural language explanations of the required functional-

| | Hand-Written | | | | Semi-Automated | | | |
|---|---|---|---|---|---|---|---|---|
| | **Time Cost** | **Line** | **Branch** | **Score** | **Time Cost** | **Line** | **Branch** | **Score** |
| **Expert_1** | 179.5 | 97.6 | 95.9 | 10.8 | 36.0 | 97.0 | 96.9 | 53.9 |
| **Expert_2** | 195.0 | 97.6 | 95.0 | 9.9 | 41.0 | 88.1 | 91.7 | 43.9 |
| **Expert_3** | 145.0 | 84.5 | 84.0 | 11.6 | 26.0 | 82.0 | 85.0 | 64.2 |
| **Expert_4** | 180.0 | 90.9 | 100.0 | 10.6 | 41.0 | 84.4 | 91.7 | 42.9 |
| **Expert_5** | 180.0 | 98.1 | 83.3 | 10.1 | 56.0 | 100.0 | 100.0 | 35.7 |
| **Total/Avg.** | 175.9 | 93.7 | 91.6 | 10.5 | 40.0 | 90.3 | 93.1 | 48.1 |

Table 4: Test case construction comparison between by Semi-Automated Pipeline and Hand-Written

ity. The landscape of code LLMs is currently experiencing a surge, with new models being introduced regularly. This includes both proprietary ones (Moradi Dakhel et al., 2023, OpenAI et al., 2023) and open-source ones (Lin, 2004, Nijkamp et al., 2023b, Touvron et al., 2023, Li et al., 2023a, Anonymous, 2024, Rozière et al., 2024), marking a trend of frequent releases in this domain.

**Code Synthesis Benchmarks.** As the capabilities of models advance, researchers are developing more challenging and versatile benchmarks for code generation. Initially, the earlier focus was on domain-specific languages (Zelle and Mooney, 1996), while the subsequent effort launched a Text-to-SQL benchmark to evaluate the capacity for generating comprehensive SQL programs (Yu et al., 2018). An investigation (Yin et al., 2018) assesses the ability to compose brief yet broadly applicable Python snippets. More recent studies (Hendrycks et al., 2021b, Li et al., 2022) have tested models' proficiency in solving competitive programming challenges using Python. A leading and extensively researched benchmark in this domain is HumanEval (Chen et al., 2021), which features 164 Python function signatures accompanied by docstrings and corresponding test cases for validating correctness. Additionally, each problem in HumanEval includes a reference solution. The MBPP (Austin et al., 2021) dataset, another Python-centric collection, was developed by having participants contribute 974 programming challenges. Each challenge encompasses a problem description (i.e., docstring), a function signature, and three test cases. There are also benchmarks for other programming languages, such as HumanEval-X (Zheng et al., 2023b) for C++, JavaScript, and Go, CodeContests (Li et al., 2022) for C++ and Java, and MultiPL-E (Cassano et al., 2022), which expands HumanEval and MBPP to 18 languages.

More recent efforts have introduced benchmarks that more closely mirror real-world coding scenarios that require interactive coding. For example, AgentBench (Liu et al., 2023b) introduces interactive tasks regarding unix shell and MySQL. SWE-Bench (Jimenez et al., 2023) compiles GitHub issues, their associated codebases, and tests, to gauge LLMs' effectiveness in practical software engineering tasks.

## 6 Conclusion

We propose NaturalCodeBench for evaluating the code generating ability of LLMs. Our benchmark comprises a total of 402 problems selected from coding online services, and it supports automatic evaluation of code generated by LLMs. We have also proposed a semi-automated pipeline for efficiently constructing the entire benchmark, achieving an efficiency gain of more than 4x compared to manual construction. We hope that NCB can provide a fair environment for the comparison between models, and our pipeline can also provide inspiration to other complex tasks or domains where evaluation costs are high.

## ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers as well as Zhipu AI for covering all GPU and API cost consumed in this study. This work is supported by The National Key Research and Development Program of China 2021YFF1201300Natural Science Foundation of China (NSFC) 62276148 and 62425601. It is also supported by Tsinghua UniversityDepartment of Computer Science and Technology-Siemens Ltd., China Joint Research Center for Industrial Intelligence and Internet of Things (JCIIOT).

## Limitations

Here, we discuss several limitations of this work.

**To cover more domains.** Although our problems are derived from real-world application scenarios, due to the difficulty of constructing accurate and efficient evaluation environments, we are unable to test some types of problems, such as those involving interface creation, web services, etc., which are also common problem types in actual applications. This results in some biases in our evaluation, which may affect the accuracy of the evaluation of certain models. We will leave these issues for future research.

**To reduce the cost.** The semi-automated pipeline can significantly reduce the time and human resources required to construct an evaluation framework, but the cost of accessing OpenAI's API remains expensive, and it does not completely eliminate the use of human resources.

## References


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*, ASE '22, New York, NY, USA. Association for Computing Machinery.

AI@Meta. 2024. Llama 3 model card.

Anonymous. 2024. Wizardcoder: Empowering code large language models with evol-instruct. In *The Twelfth International Conference on Learning Representations*.

Anthropic. 2023a. Claude-2.

Anthropic. 2023b. Introducing the claude 3 family. `https://www.anthropic.com/news/claude-3-family`. Accessed: 2024-04-28.

Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, Ramesh Nallapati, Baishakhi Ray, Parminder Bhatia, Sudipta Sengupta, Dan Roth, and Bing Xiang. 2023. Multi-lingual evaluation of code generation models. In *The Eleventh International Conference on Learning Representations*.

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.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. 2023a. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023b. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

Google AI Blog. 2024. Google gemini: Next generation model.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2022. Multipl-e: A scalable and extensible approach to benchmarking neural code generation.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya

Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2023. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113.

DeepSeek-AI. 2024. Deepseek llm: Scaling open-source language models with longtermism. *arXiv preprint arXiv:2401.02954*.

Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023a. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 423–435, New York, NY, USA. Association for Computing Machinery.

Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2023b. Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

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)*, pages 320–335, Dublin, Ireland. Association for Computational Linguistics.

Aryaz Eghbali and Michael Pradel. 2023. Crystalbleu: Precisely and efficiently measuring the similarity of code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA. Association for Computing Machinery.

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

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*.

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

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

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

Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. 2019. Code coverage at google. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 955–963, New York, NY, USA. Association for Computing Machinery.

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

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023a. Mistral 7b. *arXiv preprint arXiv:2310.06825*.

Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023b. Impact of code language models on automated program repair. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 1430–1442. IEEE Press.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

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

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-Tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. DS-1000: A natural and reliable benchmark for data science code generation. In *Proceedings of the 40th*

International Conference on Machine Learning, volume 202 of Proceedings of Machine Learning Research, pages 18319–18345. PMLR.

Raymond Li, Loubna Ben allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia LI, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Joel Lamy-Poirier, Joao Monteiro, Nicolas Gontier, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Ben Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason T Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Urvashi Bhattacharyya, Wenhao Yu, Sasha Luccioni, Paulo Villegas, Fedor Zhdanov, Tony Lee, Nadav Timor, Jennifer Ding, Claire S Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro Von Werra, and Harm de Vries. 2023a. Starcoder: may the source be with you! *Transactions on Machine Learning Research*. Reproducibility Certification.

Yuanzhi Li, Sébastien Bubeck, Ronen Eldan, Allie Del Giorno, Suriya Gunasekar, and Yin Tat Lee. 2023b. Textbooks are all you need ii: phi-1.5 technical report.

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. *Science*, 378(6624):1092–1097.

Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain. Association for Computational Linguistics.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023a. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. 2023b. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688*.

Zhe Liu, Chunyang Chen, Junjie Wang, Xing Che, Yuekai Huang, Jun Hu, and Qing Wang. 2023c. Fill in the blank: Context-aware automated text input generation for mobile gui testing. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 1355–1367. IEEE Press.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation.

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*.

Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Zhen Ming (Jack) Jiang. 2023. Github copilot ai pair programmer: Asset or liability? *J. Syst. Softw.*, 203(C).

Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023a. Codegen2: Lessons for training llms on programming and natural languages. *arXiv preprint arXiv:2305.02309*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023b. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*.

OpenAI, :, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mo Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie

Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. 2023. Gpt-4 technical report.

OpenAI. 2022. Introducing chatgpt.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems*, volume 35, pages 27730–27744. Curran Associates, Inc.

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

Phind. 2023. Phind-codellama-34b-v2.

Maja Popović. 2015. chrF: character n-gram F-score for automatic MT evaluation. In *Proceedings of the Tenth Workshop on Statistical Machine Translation*, pages 392–395, Lisbon, Portugal. Association for Computational Linguistics.

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

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20, Red Hook, NY, USA. Curran Associates Inc.

Baptiste Roziere, Jie M. Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2022. Leveraging automated unit tests for unsupervised code translation.

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

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. Llama: Open and efficient foundation language models.

Ngoc Tran, Hieu Tran, Son Nguyen, Hoan Nguyen, and Tien N. Nguyen. 2019. Does bleu score work for code migration? In *Proceedings of the 27th International Conference on Program Comprehension*, ICPC '19, page 165–176. IEEE Press.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA. Curran Associates Inc.

Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the copilots: Fusing large language models with completion engines for automated program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 172–184, New York, NY, USA. Association for Computing Machinery.

Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4all: Universal fuzzing with large language models.

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 1482–1494. IEEE Press.

Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 959–971, New York, NY, USA. Association for Computing Machinery.

Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, MAPS 2022, page 1–10, New York, NY, USA. Association for Computing Machinery.

Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. 2023a. White-box compiler fuzzing empowered by large language models.

Shuo Yang, Wei-Lin Chiang, Lianmin Zheng, Joseph E Gonzalez, and Ion Stoica. 2023b. Rethinking benchmark and contamination for language models with rephrased samples. *arXiv preprint arXiv:2311.04850*.

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 '18, page 476–486, New York, NY, USA. Association for Computing Machinery.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921, Brussels, Belgium. Association for Computational Linguistics.

John M. Zelle and Raymond J. Mooney. 1996. Learning to parse database queries using inductive logic programming. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 2*, AAAI'96, page 1050–1055. AAAI Press.

Aohan Zeng, Xiao Liu, Zhengxiao Du, Zihan Wang, Hanyu Lai, Ming Ding, Zhuoyi Yang, Yifan Xu, Wendi Zheng, Xiao Xia, Weng Lam Tam, Zixuan Ma, Yufei Xue, Jidong Zhai, Wenguang Chen, Peng Zhang, Yuxiao Dong, and Jie Tang. 2023. Glm-130b: An open bilingual pre-trained model.

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. 2023a. Judging llm-as-a-judge with mt-bench and chatbot arena.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023b. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '23, page 5673–5684, New York, NY, USA. Association for Computing Machinery.

# A Instructions in NaturalCodeBench

To enhance the efficiency of benchmark construction and reduce human labor costs, we utilized the extensive knowledge storage and natrual language understanding capabilities of LLMs during the benchmark construction process. Below are the details of the instructions used in the construction process:

- Figure 3 shows the instruction we employed to swiftly filter out queries unsuitable for testing.
- Figure 13 shows how we instruct the GPT-4 to generate diverse and high-quality testcases.
- Figure 4 illustrates how we address the issue of misalignment between class or function names generated by the LLMs and the names in the test cases.

```
I will give you a #Given Prompt# which ask the LLM to generate
code. Please verify whether the #Given Prompt# satisfies the
following requirements:
1. #Given Prompt# should contain a task, that is, the user asks the
model to help solve one or some problems.
2. It is easily to find the type of input and ouput in the #Given
Prompt#
3. There is no randomness or uncertainty in the #Given Prompt#
If the #Given Prompt# satisfies the above requirements, reply
"yes", otherwise reply "no". YOU CAN ONLY GENERATE "yes" or
"no", OTHER TOKENS ARE NOT ALLOWED.

#Given Prompt#:
{{given_prompt}}

#Response#:
```

Figure 3: The instruction used to quickly filter out low-quality queries

```
Your task is to generate {{language}} code to solve the
following problem. The generated code must be
placed between the ```{{language}} and ```, and only
one code block is allowed:
{{prompt}}

You need to follow the function names or class names
in the test cases. The generated code should not
contain any test cases:
{{test_demo}}
```

Figure 4: The instruction used to align the names of classes or functions generated by the LLMs with the names in the test cases.

# B Examples

## B.1 Examples of Semi-Automated Pipeline

In this section, we present two examples, one each for Python and Java, of semi-automated pipeline with one test case to illustrate how we construct test cases and rectify errors therein.

Figure 5 shows the Python example. Following the provision of problem description and reference solution, GPT-4 writes the majority of the test case, including the execution procedure and test case input. However, GPT-4 could not guarantee the correctness of each test case, resulting in the generation of erroneous expected outputs. At this point, our programming experts only needed to correct the incorrect expected outputs.

Figure 6 shows the Java exmaple. In this problem, where the input type involves more complex file formats, our semi-automatic pipeline is unable to directly generate the input files corresponding to each test case. Therefore, in this instance, our programming experts need to not only supplement the missing procedures in the test cases but also create an input file for each test case. However, GPT-4 has provided reference content for the input files in the comments, so our programming experts do not need to design the inputs themselves.

## B.2 Example Problems

Here, we present an example problem and test cases for each of the 6 domains.

Figure 7 shows a problem of Algorithm and Data Structure, querying the pattern of a sequence transformation and the total number of all transformations.

Figure 8 illustrates an example problem in software engineering, requiring the addition of tags to different titles in a markdown file according to their levels.

Figure 9 presents an example problem in data science, asking to select the row with the highest temperature from the temperature CSV files of each city and write these rows into a new CSV file.

Figure 10 depicts an example problem in front-end development, requiring the replacement of given special tags within a string with specific HTML formats.

Figure 11 shows an example problem in artificial intelligence, requiring the calculation of the distance between each point of two tensors, where the dimension of each tensor is batchsize * n * 3, with the third dimension representing the coordinates of the points.

Figure 12 presents an example problem in system administration, inquiring how to rename all the files within a folder according to a given rule.

| Model | Dataset | NCB(zh) | | | | NCB(en) | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Python | | Java | | Python | | Java | |
| | | Pass@10 | Pass@50 | Pass@10 | Pass@50 | Pass@10 | Pass@50 | Pass@10 | Pass@50 |
| **GPT-4** (OpenAI et al., 2023) | Test | 62.4 | 67.9 | 64.6 | 71.8 | 65.3 | 70.2 | 62.7 | 67.9 |
| | Dev | 53.3 | 55.7 | 69.2 | 72.9 | 51.8 | 54.3 | 62.0 | 64.3 |
| **GPT-3.5-Turbo** (OpenAI, 2022) | Test | 46.5 | 48.9 | 49.3 | 56.5 | 53.5 | 55.7 | 51.5 | 57.3 |
| | Dev | 44.0 | 47.7 | 45.5 | 51.4 | 43.6 | 47.1 | 48.4 | 50.0 |
| **Deepseek-Coder-33B-Instruct** (Guo et al., 2024) | Test | 55.7 | 61.8 | 48.0 | 51.1 | 56.6 | 64.9 | 52.8 | 59.5 |
| | Dev | 48.1 | 51.4 | 46.8 | 51.4 | 46.5 | 48.6 | 46.7 | 50.0 |
| **Codellama-70B-Instruct** (Roziere et al., 2023) | Test | 49.6 | 56.5 | 52.7 | 61.8 | 51.0 | 62.6 | 48.2 | 58.0 |
| | Dev | 47.5 | 54.3 | 53.9 | 62.9 | 47.6 | 54.3 | 50.5 | 60.0 |
| **Phind-Codellama-34B** (Phind, 2023) | Test | 42.3 | 46.6 | 39.4 | 45.8 | 40.6 | 43.5 | 47.6 | 56.5 |
| | Dev | 45.4 | 50.0 | 41.7 | 45.7 | 44.0 | 45.7 | 49.4 | 51.4 |
| **Deepseek-67B-Chat** (DeepSeek-AI, 2024) | Test | 44.3 | 48.9 | 40.8 | 47.8 | 47.3 | 51.9 | 40.9 | 45.8 |
| | Dev | 42.3 | 47.1 | 44.5 | 47.1 | 37.9 | 41.4 | 43.6 | 50.0 |
| **Qwen-72B-Chat** (Bai et al., 2023b) | Test | 34.9 | 37.4 | 36.5 | 39.7 | 32.7 | 35.9 | 36.5 | 38.2 |
| | Dev | 43.4 | 47.1 | 31.4 | 38.6 | 41.0 | 44.3 | 31.5 | 35.7 |
| **StarCoder** (Li et al., 2023a) | Test | 23.1 | 28.2 | 23.3 | 29.8 | 24.1 | 31.3 | 26.8 | 32.1 |
| | Dev | 29 | 32.9 | 27.3 | 32.9 | 35.5 | 41.4 | 27.0 | 30.0 |
| **Mistral-7B-Instruct** (Jiang et al., 2023a) | Test | 15.5 | 18.3 | 17.3 | 20.6 | 19.6 | 22.9 | 22.0 | 24.4 |
| | Dev | 18.2 | 21.4 | 16.3 | 20.0 | 19.7 | 24.3 | 17.8 | 21.4 |
| **CodeGen2-16B** (Nijkamp et al., 2023a) | Test | 8.6 | 16.8 | 18.0 | 22.9 | 13.0 | 19.1 | 21.0 | 26.0 |
| | Dev | 11.6 | 21.4 | 12.8 | 15.7 | 16.0 | 24.3 | 18.5 | 24.3 |
| **CodeGen-16B** (Nijkamp et al., 2023b) | Test | 4.6 | 9.2 | 13.3 | 18.3 | 9.9 | 15.3 | 17.5 | 21.4 |
| | Dev | 10.7 | 17.1 | 15.6 | 18.6 | 16.1 | 22.9 | 17.4 | 21.4 |
| **Phi-2** (Li et al., 2023b) | Test | 14.5 | 21.4 | 5.5 | 7.6 | 11.9 | 19.8 | 10.7 | 14.5 |
| | Dev | 15.3 | 27.1 | 5.1 | 7.1 | 10.9 | 18.6 | 6.4 | 7.1 |

Table 5: Pass@k results of best-performing LLMs with each LLM family on NaturalCodeBench.

## C Extra Results

Table 6 shows the pass@1 results on the development set of NCB. The results on the development set are essentially consistent with those on the test set, with some changes in the ranking among several models. This is due to differences in the distribution of problems across domains between the development set and the test set.

Table 5 shows the pass@k results of best-performing LLMs with each LLM family on NCB, where $k \in \{10, 50\}$. We do not evaluate the performance on pass@k for ErnieBot4, CodeGeeX3, Claude-3, Gemini-1.5-Pro and Llama-3-Instruct due to limitations on the use of API and other resources.

| Model | Size | NCB(zh) | | | NCB(en) | | | Total |
|---|---|---|---|---|---|---|---|---|
| | | Python | Java | Total | Python | Java | Total | |
| API LLMs | | | | | | | | |
| **GPT-4** (OpenAI et al., 2023) | N/A | 50.0 | 64.3 | 57.2 | 47.1 | 57.1 | 52.1 | 54.6 |
| **GPT-4-Turbo-1106** (OpenAI et al., 2023) | N/A | 54.3 | 55.7 | 55.0 | 50.0 | 54.3 | 52.2 | 53.6 |
| **GPT-4-Turbo-0125** (OpenAI et al., 2023) | N/A | 51.5 | 55.7 | 53.6 | 48.6 | 51.4 | 50.0 | 51.8 |
| **GPT-3.5-Turbo** (OpenAI, 2022) | N/A | 38.6 | 38.6 | 38.6 | 37.1 | 41.4 | 39.3 | 38.9 |
| **Claude-3-Opus** (Anthropic, 2023b) | N/A | 46.4 | 44.3 | 45.3 | 50.0 | 47.1 | 48.6 | 47.0 |
| **Claude-3-Haiku** (Anthropic, 2023b) | N/A | 40.3 | 32.9 | 36.6 | 43.8 | 32.9 | 38.4 | 37.5 |
| **Claude-3-Sonnet** (Anthropic, 2023b) | N/A | 37.8 | 41.4 | 39.6 | 38.6 | 31.4 | 35.0 | 37.3 |
| **Claude-2.1** (Anthropic, 2023a) | N/A | 41.4 | 37.1 | 39.3 | 35.7 | 35.7 | 35.7 | 37.5 |
| **GLM-4** (Zeng et al., 2023; Du et al., 2022) | N/A | 42.9 | 47.1 | 45.0 | 44.3 | 42.9 | 43.6 | 44.3 |
| **Gemini-1.5-Pro** (Blog, 2024) | N/A | 44.3 | 35.7 | 40.0 | 48.6 | 34.3 | 41.4 | 40.7 |
| **CodeGeeX3** (Zheng et al., 2023b) | N/A | 40.0 | 25.7 | 32.9 | 35.7 | 25.7 | 30.7 | 31.8 |
| Open LLMs | | | | | | | | |
| **Deepseek-Coder-Instruct** (Guo et al., 2024) | 33B | 41.4 | 40.0 | 40.7 | 35.7 | 41.4 | 38.6 | 39.6 |
| | 6.7B | 34.3 | 40.0 | 37.2 | 34.4 | 40.0 | 37.2 | 37.2 |
| | 1.3B | 22.9 | 21.4 | 22.2 | 20.0 | 27.1 | 23.6 | 22.9 |
| **Llama-3-Instruct** (AI@Meta, 2024) | 70B | 42.9 | 37.1 | 40.0 | 37.1 | 41.4 | 39.3 | 39.6 |
| | 8B | 22.9 | 20.0 | 21.4 | 12.9 | 20.0 | 16.4 | 18.9 |
| **Phind-Codellama** (Phind, 2023) | 34B | 34.1 | 31.4 | 32.8 | 38.6 | 40.0 | 39.3 | 36.0 |
| **Qwen-1.5** (Bai et al., 2023a) | 110B | 35.7 | 30.0 | 32.9 | 37.1 | 35.7 | 36.4 | 34.6 |
| **Codellama-Instruct** (Roziere et al., 2023) | 70B | 30.0 | 30.0 | 30.0 | 35.7 | 35.7 | 35.7 | 32.9 |
| | 34B | 14.3 | 25.7 | 20.0 | 25.7 | 25.7 | 25.7 | 22.9 |
| | 13B | 21.4 | 20.0 | 20.7 | 22.9 | 20.0 | 21.5 | 21.1 |
| | 7B | 25.7 | 14.3 | 20.0 | 18.6 | 17.1 | 17.9 | 18.9 |
| **Deepseek-Chat** (DeepSeek-AI, 2024) | 67B | 28.6 | 35.7 | 32.2 | 28.6 | 32.9 | 30.8 | 31.5 |
| | 7B | 12.9 | 11.4 | 12.2 | 10.0 | 14.3 | 12.2 | 12.2 |
| **WizardCoder** (Luo et al., 2023) | 34B | 31.4 | 31.4 | 31.4 | 30.0 | 31.4 | 30.7 | 31.1 |
| | 15B | 30.0 | 24.3 | 27.2 | 31.4 | 24.3 | 27.9 | 27.5 |
| **Qwen-Chat** (Bai et al., 2023b) | 72B | 35.7 | 24.3 | 30.0 | 34.3 | 25.7 | 30.0 | 30.0 |
| | 7B | 10.0 | 12.9 | 11.5 | 20.0 | 15.7 | 17.9 | 14.7 |
| **StarCoder** (Li et al., 2023a) | 15.5B | 17.1 | 15.7 | 16.4 | 21.4 | 15.7 | 18.6 | 17.5 |
| **Mistral-Instruct** (Jiang et al., 2023a) | 7B | 11.4 | 12.9 | 12.2 | 15.7 | 11.4 | 13.6 | 12.9 |
| **CodeGen2** (Nijkamp et al., 2023a) | 16B | 5.7 | 7.1 | 6.4 | 8.6 | 7.1 | 7.9 | 7.1 |
| | 7B | 1.4 | 5.7 | 3.6 | 1.4 | 5.7 | 3.6 | 3.6 |
| | 3.7B | 0.0 | 5.7 | 2.9 | 2.9 | 2.9 | 2.9 | 2.9 |
| | 1B | 0.0 | 2.9 | 1.5 | 0.0 | 2.9 | 1.5 | 1.5 |
| **CodeGen** (Nijkamp et al., 2023b) | 16B | 1.4 | 5.7 | 3.6 | 7.1 | 8.6 | 8.6 | 5.7 |
| | 6B | 2.9 | 2.9 | 2.9 | 4.3 | 7.1 | 5.7 | 4.3 |
| | 2B | 0.0 | 2.9 | 1.5 | 2.9 | 5.7 | 4.3 | 2.9 |
| **Phi** (Li et al., 2023b) | 2.7B | 4.3 | 4.3 | 4.3 | 5.7 | 4.3 | 5.0 | 4.7 |
| | 1.3B | 1.4 | 2.9 | 2.2 | 5.7 | 4.3 | 5.0 | 3.6 |

Table 6: Evaluating LLMs on the dev set of NaturalCodeBench. All results are pass@1 on greedy decoding.

***Problem***

*I have a dataframe that includes the price and date of a symbol, how can I identify the time periods where the price has consistently fluctuated within an x percent range?*
*For instance, the output of the following statements:*
*1) From December 10 to December 30*
*2) From March 10 to March 23*

**Reference Solution**

```python
def find_fluctuation_periods(df, symbol, x):
    symbol_data = df[…==symbol].sort_values(by='date')
    …
    for ind, row in symbol_data.iterrows():
        if start_date is None:
            …
        else:
            change = abs((row['price'] - prev_price) / prev_price
* 100)
            if change > x:
                if ind - start_ind > 1:
                    periods.append((start_date.strftime('%Y-%m-
%d'), prev_date.strftime('%Y-%m-%d')))
                …
    if start_date != end_date:
        periods.append((start_date.strftime('%Y-%m-%d'),
end_date.strftime('%Y-%m-%d')))
    return periods
```

**Test Case Generated by GPT-4**

```python
def test_fluctuation_periods_2(self):
    df = pd.DataFrame({
        'symbol': ['AAPL', 'AAPL', 'AAPL', 'AAPL'],
        'price': [100, 110, 120, 130],
        'date': pd.to_datetime([
            '2021-01-01',
            '2021-01-02',
            '2021-01-03',
            '2021-01-04'])
    })
    assert find_fluctuation_periods(df, 'AAPL', 10) ==
[('2021-01-01', '2021-01-03')]
```

**Wrong Output**

**Human Rewritten Test Case**

```python
def test_fluctuation_periods_2(self):
    df = pd.DataFrame({
        'symbol': ['AAPL', 'AAPL', 'AAPL', 'AAPL'],
        'price': [100, 110, 120, 130],
        'date': pd.to_datetime([
            '2021-01-01',
            '2021-01-02',
            '2021-01-03',
            '2021-01-04'])
    })
    assert find_fluctuation_periods(df, 'AAPL', 10) ==
[('2021-01-01', '2021-01-04')]
```

Figure 5: A Python example of semi-automate pipeline.

***Problem***

*Design a method in Java*
*Use the following encryption method, encrypt the content in the given encodingFile text file, and then save it to the encodedFile file.*
*Encryption rules:*
*1. Numbers: If it is not the number 9, add 1 to the original basis, If it is the number 9, it becomes 0.*
*2. Letter characters: If it is a non-z character, move one to the right, If it is z, z->a, Z->A.*
*3. Non-numeric and non-letter characters can remain unchanged, such as Chinese characters and punctuation marks, etc., just need to remain unchanged.*

**Reference Solution**

```java
void encodeFile(File encodingFile, File encodedFile) {
    try (FileReader reader = …(encodingFile);
        FileWriter writer = …(encodedFile)) {
        while ((c = reader.read()) != -1) {
            char character = (char) c;
            if (Character.isDigit(character)) {
                character = character == '9' ? '0' : (char)
(character + 1);
            }else if (Character.isLetter(character)) {
                . . .
                else if ((character >= 'a' && …) {
                    character=(char)(character+1);
  . . .
```

**Test Case Generated by GPT-4**

```java
@Test
void testEncodeDigits() throws IOException {
    File input = new File("testEncode.txt");
    File output = new File("testEncodeOutput.txt");
    // numbers.txt contains "123456789"
    // encodedNumbers.txt should contain
"234567890"
}
```

**Not completely generated**

**Human Rewritten Test Case**

```java
@Test
void testEncodeDigits() throws IOException {
    File input = new File("testEncode.txt");
    File output = new File("testEncodeOutput.txt");
    FileEncoder.encodeFile(input, output);
    assertEquals("234567890",
        readFileContent(output));
}
```

Figure 6: A Java example of semi-automate pipeline.

*Problem:*

*Given a sequence that only contains two possible characters "O" and "x". There is a magical operation that can combine two consecutive "x" characters in the sequence into one "O" character. Suppose there is a sequence of length n, containing only "x" characters, and the magic operation can be used any number of times. What is the maximum number of possible result sequences?*
*For example:*
*For a sequence of length 2, the initial state is "xx", you can choose not to use the magic operation or use it only once. There are two possible final results: "xX" or "O".*
*For a sequence of length 3, the initial state is "xxX", you can choose not to use the magic operation or use it once. There are three possible final results: "xxx", "OX!" (combining the first two "x" characters) or "XO" (combining the last two "x" characters).*

**Rerference Solution**

```
def max_possible_sequences(n):
    if n <= 0:
        return 0
    elif n == 0:
        return 0
    elif n == 0:
        return 2
    else:
        return max_possible_sequences(n-1) \
          + max_possible_sequences(n-1)
```

**Test Cases**

```
class Testmax_possible_sequences:
    def test_max_possible_sequences_1(self):
        assert max_possible_sequences(4) == 5
    def test_max_possible_sequences_2(self):
        assert max_possible_sequences(7) == 21
```
...

Figure 7: An example problem of Algorithm and Data Structure.

*Problem:*

*Hello, please write a Python function for me. The function should read a markdown file, add numbering like x.y.z... to the titles of each level, and then return the modified string. Please note not to write into the original file.*

**Test Cases**

```
class Testadd_section_numbering:
    def test_case1(self):
    with open('test1.md', 'w') as f:
        f.write('# Title\n## Subtitle\n### Sub-Subtitle\n##
Another Subtitle\n# Another Title')
    assert add_section_numbering(
        'test1.md') == '# 1 Title\n## 1.1 Subtitle\n### 1.1.1
Sub-Subtitle\n## 1.2 Another Subtitle\n# 2 Another Title'
```
...

**Rerference Solution**

```
def add_section_numbering(markdown_file):
    with open(markdown_file, 'r') as file:
        lines = file.readlines()

    numbering = []
    result = ''
    for line in lines:
        if line.startswith('#'):
            level = line.count('#')
            numbering = numbering[:level]
            if len(numbering) < level:
                numbering.append(0)
            numbering[-1] += 1
            line = '#'*level + ' ' + '.'.join(map(str, numbering))
+ ' ' + line[level:].strip() + '\n'
        result += line
    return result[:-1]
```

Figure 8: An example problem of Software Engineering.

*Problem:*

*There are multiple CSV files in the data folder, each file has two columns, containing the daily temperature records of a certain city in 2022. The first row is the title, which are Date and Temperature. The temperature value is an integer. I need to find out the highest temperature value and the corresponding date of each city in that year, and save the results to a new CSV file. The result CSV consists of three columns, including city, highest temperature, and date. Note that if the highest temperature is the same for multiple days, keep all dates that reach the highest temperature. How can I use the pandas library's dataframe to complete this task?*

**Test Cases**

```
class Testmax_possible_sequences:
    def test_single_file_single_max(self, tmpdir):
        data =
"Date,Temperature\n2022-01-01,10\n2022-01-02,20\n2022-01-
03,30"
        p = tmpdir.mkdir("data").join("city1.csv")
        p.write(data)
        output_file = tmpdir.join("output.csv")
       find_max_temperature(str(tmpdir.join("data")),
str(output_file))
        assert output_file.read() ==
"City,Max_Temperature,Date\ncity1,30,2022-01-03\n"
                                ...
```

**Rerference Solution**

```
def find_max_temperature(folder_path, output_file):
    csv_files = [f for f in os.listdir(folder_path)
        if f.endswith('.csv')]
    result_df = pd.DataFrame(columns=[
        'City',
        'Max_Temperature',
        'Date'])
    for csv_file in csv_files:
        file_path = os.path.join(folder_path, csv_file)
        df = pd.read_csv(file_path)
        city_name = csv_file[:-4]
        max_temp = df['Temperature'].max()
        max_temp_dates = df.loc[
            df['Temperature'] == max_temp,
            'Date'].tolist()
        for date in max_temp_dates:
            result_df = result_df._append({
                'City': city_name,
                'Max_Temperature': max_temp,
                'Date': date}, ignore_index=True)
    result_df.to_csv(output_file, index=False)
```

Figure 9: An example problem of Data Science.

*Problem:*

*How to replace a string containing content like ```html ```, ```css ```, ```python ```, ```javascript ```, ```golang ``` with strings like <pre><code class=\"language-html\">...</code></pre>, <pre><code class=\"language-css\">...</code></pre>, <pre><code class=\"language-python\">...</code></pre>, <pre><code class=\"language-javascript\">...</code></pre>, <pre><code class=\"language-golang\">...</code></pre>. Please use python code.*

**Test Cases**

````
class Testreplace_code_block:
    def test_replace_code_block_1(self):
        assert replace_code_block('```html ```') == '<pre><code
class="language-html"></code></pre>'

                                ...
````

**Rerference Solution**

````
def replace_code_block(text):
    languages = {
        "html": "language-html",
        "css": "language-css",
        "python": "language-python",
        "javascript": "language-javascript",
        "golang": "language-golang"
    }
    for lang, html_class in languages.items():
        pattern = rf"```{lang}\b\s*(.*?)\s*```"
        replacement = rf'<pre><code
class="{html_class}">\1</code></pre>'
        text = re.sub(pattern, replacement, text,
flags=re.DOTALL)
    return text
````

Figure 10: An example problem of Front-End.

*Problem:*

*Python code, calculate distance given two Pytorch tensors with dimension batchsize x n x 3, n is points, 3 is x,y,z. Compute point wise distance along the last dimension, for example only compute distance between a[0,1] and b[0,1] not a[0,1] and b[0,2].*

**Rerference Solution**

```
def calculate_distance(tensor_a, tensor_b):
    diff = tensor_a - tensor_b
    dist = torch.sqrt(torch.sum(diff ** 2, dim=-1))
    return dist
```

**Test Cases**

```
class Testcalculate_distance:
    def test_case_1(self):
        tensor_a = torch.tensor([[[1,2,3],[4,5,6]]])
        tensor_b = torch.tensor([[[1,2,3],[4,5,6]]])
        expected_output = torch.tensor([[0.0, 0.0]])
        assert torch.allclose(calculate_distance(tensor_a,
tensor_b), expected_output)

    def test_case_2(self):
        tensor_a = torch.tensor([[[1,1,1],[2,2,2]]])
        tensor_b = torch.tensor([[[0,0,0],[0,0,0]]])
        expected_output = torch.tensor([[1.7321,
3.4641]])
        assert torch.allclose(calculate_distance(tensor_a,
tensor_b), expected_output, atol=1e-4)

...
```

Figure 11: An example problem of Artificial Intelligence.

*Problem:*

*I want to write a python program that rename the files of a folder .*
*please remove all letters and keep the numbers*

**Rerference Solution**

```
def rename_files_in_folder(folder_path):
    for filename in os.listdir(folder_path):
        file_type = filename.split('.')[-1]
        new_filename = re.sub("[A-Za-z]", "",
filename[:-len(file_type)]) + file_type
        os.rename(os.path.join(folder_path, filename),
os.path.join(folder_path, new_filename))
```

**Test Cases**

```
class Testrename_files_in_folder:
    def test_rename_files_in_folder_1(self, tmpdir):
        p = tmpdir.mkdir("sub").join("file123abc.txt")
        p.write("content")
        rename_files_in_folder(str(tmpdir) + '/sub/')
        assert os.path.isfile(str(tmpdir) + '/sub/123.txt')

    def test_rename_files_in_folder_2(self, tmpdir):
        p = tmpdir.mkdir("sub").join("file456def.txt")
        p.write("content")
        rename_files_in_folder(str(tmpdir) + '/sub/')
        assert os.path.isfile(str(tmpdir) + '/sub/
456.txt')

    def test_rename_files_in_folder_3(self, tmpdir):
        p = tmpdir.mkdir("sub").join("file789ghi.txt")
        p.write("content")
        rename_files_in_folder(str(tmpdir) + '/sub/')
        assert os.path.isfile(str(tmpdir) + '/sub/
789.txt')
...
```

Figure 12: An example problem of System Administration.

*I will give you a #Prompt# and a piece of #Code#. I need you to write 10 diverse test cases to verify whether the function in the #Code# meets the requirements of the #Prompt#. Among them, 6 test cases should cover as many lines and branches in the #Code# as possible, and the other 4 test cases should try to reach the boundaries of the requirements in the #Prompt#. The test cases should conform to the Pytest/JUnit call format. You should only generate test cases without any explanation.*

```
#Prompt#:
{{given_prompt}}

#Code#:
```
{{given_code}}
```


#Test cases#:
class Test{{class_name}} :/{
```

Figure 13: The insturciton used in Semi-automated Pipeline. Generating 6 test cases for high-coverage and 4 corner test cases.