Review



# Assessing Code Reasoning in Large Language Models: A Literature Review of Benchmarks and Future Directions

Saman Dehghan *

Posted Date: 15 November 2024

doi: 10.20944/preprints202411.1147.v1

Keywords: Large Language Models (LLMs); Code Reasoning; Software Development; Code Comprehension

 doi:10.20944/preprints202411.1147.v1



*Article*

# Assessing Code Reasoning in Large Language Models: A Literature Review of Benchmarks and Future Directions

***Code Reasoning* Benchmarks in LLMs**

**Saman Dehghan**

Department of Computer Science, IL, USA; saman@enumclass.cc

**Abstract:** The growing role of Large Language Models (LLMs) in software development calls for robust methods to evaluate their code reasoning capabilities. Although LLMs perform well in code generation, accurately assessing their deeper understanding of programming logic, execution behavior, and semantic analysis remains a key challenge. This paper presents a comprehensive review of benchmarks and evaluation frameworks designed to measure code reasoning in LLMs. We explore key terminology, concepts, and frameworks for evaluating code reasoning in LLMs and then analyze the limitations of traditional benchmarks. Emerging evaluation frameworks that assess deeper comprehension through code interpretation, execution behavior, and semantic understanding are also examined. By identifying current gaps and challenges, we offer insights into the state of code reasoning evaluation and propose directions for future research to develop more robust, unbiased benchmarks. This review aims to guide researchers and practitioners in selecting effective evaluation frameworks and advancing the development of LLMs with enhanced code reasoning capabilities.

**Keywords:** large language models (LLMs); code reasoning; software development; code comprehension

---

## 1. Introduction

The emergence of Large Language Models (LLMs) has transformed the landscape of artificial intelligence, particularly in code understanding and generation. These models have demonstrated remarkable capabilities in tasks ranging from code completion to complex program analysis, leading to their increasing adoption in software development workflows. However, accurately assessing these models' true code reasoning capabilities beyond mere code generation remains a critical challenge.
Code reasoning involves understanding code semantics, predicting program behavior, and executing or simulating code snippets. Unlike traditional code generation and completion tasks, code reasoning requires a deeper comprehension of programming logic, control flow, and data structures. Evaluating LLMs on these aspects poses unique challenges due to the complexity and diversity of programming problems. To address these challenges, various benchmarks have been proposed, each aiming to measure different facets of code reasoning in LLMs.
On the other hand, Traditional evaluation approaches have primarily focused on code generation tasks, measuring a model's ability to translate natural language descriptions into executable code. However, this narrow focus fails to capture the complexity of real-world programming tasks, where understanding existing code, debugging issues, and reasoning about program behavior are essential skills. The limitations of these traditional benchmarks have become increasingly apparent as LLMs advance, revealing gaps in our ability to comprehensively assess their code reasoning capabilities.
This paper addresses these challenges by providing a systematic review of code reasoning evaluation methodologies. We begin by establishing a foundational framework of terminology and concepts essential for understanding code reasoning in LLMs. Our analysis encompasses both traditional evaluation metrics and emerging frameworks, examining their effectiveness in assessing different aspects of code comprehension and reasoning.
We identify five key dimensions that characterize the evolution of code reasoning evaluation

1. The transition from simple generation tasks to complex reasoning assessment

 doi:10.20944/preprints202411.1147.v1



2. The incorporation of execution behavior and runtime analysis
3. The development of contamination-resistant evaluation frameworks
4. The expansion to multilingual code understanding
5. The integration of real-world programming scenarios

Through this comprehensive analysis, we reveal critical gaps in current evaluation methodologies and propose directions for future research. Our findings highlight the need for more sophisticated evaluation frameworks that can accurately assess an LLM's ability to understand and reason about code across different programming languages, paradigms, and complexity levels. This review makes several contributions to the field:

- Provides a structured analysis of existing code reasoning evaluation methodologies
- Identifies limitations and potential biases in current evaluation approaches
- Offers insights into designing more robust and comprehensive evaluation frameworks
- Proposes future directions for advancing code reasoning assessment

Understanding these aspects is crucial for researchers and practitioners who need to select appropriate evaluation methods for their specific use cases. Moreover, this knowledge is essential for developing the next generation of more robust and comprehensive benchmarks.

As the field of AI-assisted programming continues to evolve rapidly, having well-designed and uncontaminated evaluation frameworks becomes increasingly important. These benchmarks not only help measure progress but also guide the development of more capable and reliable models. By providing a comprehensive overview of existing approaches, this review seeks to facilitate better understanding and advancement of code reasoning capabilities in LLMs.

Existing literature on benchmarks and evaluation methods for Large Language Models (LLMs) often lacks a focused on code reasoning evaluation, which is the aim of our paper. In [1], the authors provide a comprehensive review of methods and metrics used in code generation tasks, such as functional correctness and syntactic quality. However, the study primarily emphasizes these metrics from a general performance standpoint rather than addressing the specific challenges associated with code reasoning. A panoramic perspective on LLM evaluation is offered in [2], where the authors outline diverse evaluation criteria across multiple dimensions. Nevertheless, the scope is broad and lacks a deep dive into benchmarks tailored specifically for code reasoning. Similarly, in [3], the authors categorize evaluation methods into what to evaluate, where to evaluate, and how to evaluate, presenting a systematic framework that remains more focused on general LLM tasks rather than code reasoning.The study in [4] explores automated benchmarking frameworks for assessing computer science concept inventories, yet it emphasizes automation in education rather than the distinct evaluation needs for code reasoning. Finally, [5] surveys prevalent methodologies to evaluate the reasoning behavior of LLMs, highlighting trends toward nuanced reasoning analyses but not specifically addressing the benchmarks and methods for evaluating code reasoning. Our work stands out by offering a clear and targeted review of benchmarks and evaluation methods for code reasoning in LLMs, aimed at filling these gaps and presenting future directions that are accessible and highly relevant to researchers in this field.

The remainder of this paper is organized as follows: Section 2 establishes the fundamental terminology and concepts in code reasoning evaluation. Section 3 analyzes traditional code generation benchmarks and their limitations. Section 4 introduces emerging code reasoning benchmarks and their novel approaches to evaluation. Section 5 provides future directions for Code Reasoning Benchmarks and Finally, Section 6 concludes with a discussion of current challenges in code reasoning evaluation.

## 2. Terminology and Key Concepts

This section establishes the fundamental terminology and concepts used throughout our review of code reasoning evaluation in Large Language Models (LLMs).

 doi:10.20944/preprints202411.1147.v1



### *2.1. Reasoning*

Reasoning is the mental process of reaching logical conclusions based on the information available[5]. When we talk about reasoning in Large Language Models (LLMs), it involves several steps:

- **Analyzing the input data**: Similar to how humans break down information, LLMs must first analyze the input they are given.
- **Identifying patterns**: LLMs excel at identifying patterns in data, and in the context of code, this could involve recognizing common programming constructs, syntax rules, or relationships between variables.
- **Making inferences**: Based on the patterns they identify, LLMs draw conclusions and make inferences about the code's behavior or functionality.
- **Formulating coherent and logically consistent outputs**: The goal is for LLMs to generate outputs, such as code completions, program repairs, or explanations, that are not only correct but also make logical sense within the given context.

LLMs are expected to perform two primary types of reasoning:

- **Deductive reasoning**: This involves drawing specific conclusions from general rules or principles. For example, if an LLM knows that all functions must have a return statement (general principle) and it encounters a function without one, it can be deduced that the code is likely incorrect.
- **Inductive reasoning**: This is the process of inferring general rules from specific examples. For example, if an LLM observes many examples of loops being used to iterate over arrays, it might induce a general rule about the purpose and structure of loops.

#### 2.1.1. Reasoning Behavior

Reasoning behavior can be understood as the visible responses and actions of a system as it tackles a reasoning problem[5]. It's not just about the end result; it encompasses the entire process, including the intermediate steps, methods, and strategies employed by the model. Observing reasoning behavior helps us grasp:

- **Input Interpretation**: How does the model understand and process the information it's given?
- **Application of Logic**: What logical operations does the model use, and how does it apply them to reach conclusions?
- **Conclusion Generation**: How does the model arrive at its final output, and what does that reveal about its internal workings and patterns of "thought"?

Examining reasoning behavior provides valuable insights into the mechanisms driving a system's reasoning capabilities.

#### 2.1.2. Code Reasoning

Code reasoning [6] is the capacity of models, specifically Large Language Models (LLMs), to comprehend, analyze, and reason about code and algorithms. It is a multifaceted ability encompassing several key aspects:

- **Code Analysis**: This involves examining code to grasp its syntax, semantics, and functionality. LLMs need to identify programming constructs, understand the meaning of code elements, and trace the flow of execution to understand how the code works
- **Question Answering**: Code reasoning includes responding to questions about code behavior, logic, and outcomes. This can involve multiple-choice questions, where the model selects the correct answer from a set of options, or free-form question answering, where the model generates a textual response explaining the code's behavior.
- **Conceptual Understanding**: LLMs must exhibit knowledge of fundamental programming concepts, data structures, and algorithms. This understanding allows them to reason about the higher-level purpose and design of code, rather than just its syntactic details.

 doi:10.20944/preprints202411.1147.v1



- **Numerical Calculation**: Code often involves numerical computations. LLMs need to perform these calculations accurately to understand the program's state and predict its output.
- **Complexity Analysis**: Assessing the time and space complexity of algorithms is a crucial aspect of code reasoning. LLMs should be able to analyze how the resources required by an algorithm scale with the input size.
- **Algorithm Behavior Prediction**: Given specific inputs or conditions, LLMs should be able to predict the output or behavior of the code. This requires understanding the algorithm's logic and how it operates on the input data.

#### 2.1.3. Coding for Reasoning

Coding for reasoning [6] is a specialized approach where LLMs generate code as a tool to solve complex problems, particularly those involving intricate calculations or repetitive steps (iterative processes). This method allows models to leverage their code generation capabilities to enhance their reasoning abilities in several ways:

- **Generate Code Solutions**: Write code snippets or programs that implement algorithms to solve given problems.
- **Leverage External Execution**: Utilize interpreters or compilers to execute the generated code and obtain results, especially for tasks that are computationally intensive.
- **Abstract Complex Reasoning**: Simplify complex reasoning tasks by translating them into code, thereby offloading computational aspects to code execution.
- **Internal Code Simulation**: Perform mental simulations of code execution to enhance reasoning without actual external execution, aiding in problem-solving and verification.

### *2.2. Evaluation Metrics*

To effectively assess the performance of LLMs in code reasoning and generation tasks, a variety of evaluation metrics are employed, each focusing on different aspects of the generated code.

#### 2.2.1. Syntactic Correctness

Syntactic correctness is a fundamental measure of code quality, assessing the proportion of generated code that adheres to the grammatical rules of the programming language. This means the code can be successfully parsed or compiled without errors. It is a basic requirement for any executable code, serving as a foundation for further evaluation of functionality and meaning. There are some metrics include:

- **Compilation Accuracy** [7]: This metric examines the success rate of program repairs in terms of compilation, calculating the proportion of generated fixes that compile successfully. It verifies whether the produced code conforms to the target programming language's syntax specifications and can be correctly interpreted by compilers. Evaluation is performed using standard compilation tools such as javac (Java) and pycompile (Python). Achieving high Compilation Accuracy demonstrates that the generated repairs are syntactically well-formed and ready for functional validation tests.
- **EDIT-SIM** [8]: measures the editing effort needed to turn generated code into a correct solution. Even if code fails some tests, it can still be useful if minimal edits are required. It is defined as:

$$\text{EDIT-SIM} = 1 - \frac{\text{lev}(\text{gen}, \text{ref})}{\max(\text{len}(\text{gen}), \text{len}(\text{ref}))}$$

in which gen is the generated code or system output and ref is the reference correct solution and lev is the Levenshtein edit distance, which calculates the number of single-character edits needed to match gen to ref. A higher EDIT-SIM indicates less effort needed to correct the code, making it more desirable.

 doi:10.20944/preprints202411.1147.v1



To sum up Syntactic Correctness:

- **Focus on Grammar**: Syntactic correctness focuses primarily on the structure and form of the code, ensuring that it follows the specific rules of the programming language. For instance, in Python, proper indentation is crucial for defining code blocks, while in Java, semicolons are used to terminate statements.
- **Importance in Cross-Language Generalization**: As LLMs are increasingly trained on multiple programming languages, the ability to generalize syntactically across languages becomes essential. Even if a model hasn't been specifically trained on a certain language, a high syntactic correctness rate in that language indicates some level of cross-language understanding.
- **Relationship to Semantic Correctness**: While syntactic correctness is a prerequisite for executable code, it doesn't guarantee that the code will function as intended. Semantic correctness goes beyond syntax to assess whether the code actually implements the desired logic and produces the correct outputs. The sources highlight that LLMs can sometimes generate syntactically valid code that fails to pass tests due to semantic errors.

#### 2.2.2. Reference Match

Reference matching, also referred to as "exact match," assesses how closely the generated code aligns with a provided reference solution. There are several techniques within the realm of reference matching:

- **N-gram Matching**: This technique involves comparing short sequences of words or tokens (n-grams) between the generated code and the reference solution. Common examples include BLEU [9], ROUGE [10], METEOR[11], and chrF[12], often used in language-based tasks and adapted for code evaluation.
- **Code-Specific N-gram Matching**: Building upon basic n-gram matching, metrics like CodeBLEU[13], RUBY[14], and CrystalBLEU[15] incorporate code-specific structural features. They consider elements like data flow, program dependency graphs (PDGs), or abstract syntax trees (ASTs) to capture more nuanced code similarities. These metrics might also filter out frequently occurring n-grams that don't contribute significantly to the meaning of the code, thus enhancing the accuracy of the evaluation.
- **Embedding-Based Methods**: These techniques utilize advanced language models to transform both the generated code and the reference solution into vector representations (embeddings). The similarity between these vectors then serves as a measure of how closely the generated code matches the reference. Examples of embedding-based metrics include BERTScore[16] and CodeBERTScore[17]

#### 2.2.3. Functional Correctness

Functional correctness, also known as reference-free evaluation, focuses on assessing whether the generated code performs its intended task accurately, irrespective of its structural similarity to a specific reference solution. This approach prioritizes the code's behavior and ability to meet predefined criteria or pass designated unit tests as the primary measure of its quality. There are various methods for evaluating functional correctness, each with its own strengths and considerations:

- **Execution-Based Methods**: Direct Assessment Through Testing: This approach involves running the generated code on a set of test cases and comparing its output to the expected results. It stands as one of the most reliable and widely adopted methods for evaluating functional correctness. Examples of execution-based methods includes:

 doi:10.20944/preprints202411.1147.v1



  - **pass@k** [18]: measures the probability that at least one of the k generated samples for a problem solves the problem. It evaluates the likelihood that, if k code samples are generated, at least one of them will pass all test cases. It's calculated as

$$\text{pass@k} := \mathbb{E}_{\text{problems}}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right]$$

    Here k is the number of programs sampled from the model and c is the number of correct programs that pass all test cases. A higher pass@k indicates a greater likelihood that the model produces at least one functionally correct solution among the k samples.
  - **n@k** [19] (Top-n from k Samples): n@k measures the percentage of problems a model can solve using n submissions selected from k generated samples per problem. The model generates k code solutions for each problem and tests n of them. If any of these n solutions pass all test cases, the problem is considered solved. This metric shows how successful the model is when it has multiple attempts to produce a correct solution.
  - **Execution Accuracy** [20]: Measures how often the generated code runs without errors or crashes, focusing on syntax and execution rather than output correctness.
  - **Test Case Average** [21]: The average percentage of test cases passed by the generated code, allowing for partial correctness and recognizing that some solutions may handle common cases but miss edge cases.
  - **Strict Accuracy** [21]: The percentage of problems for which the generated code passes all test cases, including edge cases. This metric requires complete correctness and sets a high standard for performance.

- **LLM-based Methods**: LLM-based approaches present an innovative way to assess the functional correctness of code generated by other LLMs. This evaluation method leverages the advanced code understanding and reasoning capabilities of LLMs to analyze the generated code and offer insights into its behavior and correctness. Examples of LLM-based Methods includes ICE-Score[22], RTC[23]
- **Hybrid methods**: combine different evaluation approaches to provide a more comprehensive assessment of code generation systems. They leverage the strengths of each method while mitigating their individual weaknesses. example of hybrid methods includes CodeScore[24] which trains a language model to capture execution semantics and can utilize both the NL context and reference code.

#### 2.2.4. Human Evaluation

Involves human judgment of the generated code. While expensive, this approach can capture aspects that automated metrics might miss, such as code readability, maintainability, and adherence to coding conventions. Human evaluation can also provide insights into a developer's productivity when using a code generation model. An example of Human evaluation framework is RealHumanEval[25] framework

### Conclusion

A thorough understanding of these core concepts and evaluation metrics is essential for assessing the code reasoning capabilities of LLMs. By employing a comprehensive set of metrics that cover syntactic validity, functional correctness, efficiency, code quality, and robustness, we can gain deeper insights into the strengths and limitations of these models in code reasoning tasks. This holistic evaluation approach enables researchers and practitioners to identify areas for improvement and to advance the development of more capable and reliable code-oriented language models.

## 3. Code Generation Benchmarks

The primary method for evaluating code-generating Language Models (LLMs) has been through benchmarks such as HumanEval [26] and MBPP (Mostly Basic Python Problems) [27]. These bench-

 doi:10.20944/preprints202411.1147.v1



marks try to assess code-generating ability HumanEval and MBPP are widely used benchmarks that test LLMs by generating code from natural language specifications. Their evaluation typically focuses on:

- Generating Python code from natural language descriptions.
- Converting docstrings into executable programs.
- Testing standalone functions that are relatively simple and self-contained.

These benchmarks offer useful insights, especially in evaluating basic program synthesis, but their focus is narrow. As LLMs advance, it becomes essential to assess their abilities more comprehensively by addressing the key limitations inherent in these benchmarks.

### *3.1. Key Limitations of Traditional Benchmarks*

#### 3.1.1. Solution Accuracy Issues

Despite achieving high pass rates, many solutions deemed correct are actually flawed. HumanEval+ [28] addresses this issue by enhancing HumanEval with more rigorous test cases, revealing that numerous previously accepted solutions contained subtle bugs or logical errors [29]. This highlights a critical gap that evaluating LLMs with superficial tests may lead to overestimating their actual performance.

#### 3.1.2. Language Coverage

These benchmarks primarily evaluate Python code generation, limiting the assessment to a single programming language. Extensions like HumanEval-X [30], MultiPLe [31], and MBXP [32] have been developed to include programming languages beyond Python, but the scope remains limited.

#### 3.1.3. Limited Problem Variety and Scale

Constructed mainly by human annotators, these benchmarks contain a limited number of problems due to the high manual effort required. For instance, HumanEval [26] includes only 164 handcrafted problems, which is insufficient to fully measure the comprehensive program synthesis capabilities of modern LMs. The problems are often self-contained and lack diversity in types and domains, providing only a baseline overview without offering deeper insights into specific strengths or weaknesses of the models.

#### 3.1.4. Data Leakage and Training Dataset Composition

Popular benchmarks like HumanEval [26] and MBPP [27] were released several years ago, with example solutions available in open-source repositories. Recent LMs have achieved higher pass@1 scores, often with minimal differences between models. However, it's unclear how much of this improvement is due to leaked solutions included in training data. Studies have shown substantial overlap between benchmark solutions and open-source training corpora. Closed source LMs may even include benchmark ground truths to artificially boost their leaderboard standings. Potential sources of contamination include direct data leakage, indirect leakage through synthetic data, and over-fitting during model selection [33].
Benchmarks like LBPP [33], an uncontaminated set of 161 prompts with associated Python solutions, and LiveBench [34], which includes frequently updated questions from recent sources, aim to mitigate these issues

#### 3.1.5. Further limitation

Beyond these concerns, traditional benchmarks also face further limitations that restrict their ability to comprehensively evaluate code LMs:

 doi:10.20944/preprints202411.1147.v1



- **Emphasis on Code Generation Over Understanding**: Most benchmarks prioritize the generation of code from descriptions, overlooking the models' abilities to comprehend existing code or reason about its functionality and structure.
- **Lack of Real-World Problem Representation**: The problems are often artificial and do not reflect the complexity and diversity of real-world coding tasks, limiting the applicability of the evaluation results to practical scenarios.
- **Minimal Testing of Execution Capabilities**: There is limited assessment of a model's ability to execute code or understand its runtime behavior, which is crucial for tasks like debugging and optimization.
- **Insufficient Problem Quantity**: The small number of problems (e.g., HumanEval's 164 problems) is inadequate for a comprehensive evaluation, as it cannot cover the vast array of programming concepts and scenarios encountered in practice.
- **Inadequate Evaluation Metrics**: Reliance on binary pass/fail outcomes may overlook nuanced understanding and fail to capture partial credit for solutions that are nearly correct but may have minor errors or inefficiencies.
- **Overemphasis on Functional Correctness**: Focusing solely on whether the code works neglects other important aspects of code quality, such as readability, efficiency, maintainability, and adherence to best practices.

Conclusion

In light of these limitations, traditional code benchmarks do not fully capture the breadth and depth of a model's programming abilities. They often miss evaluating the critical skills required for understanding and reasoning about code, which are essential for real-world programming tasks. To address these gaps, we will focus on code reasoning benchmarks in the following sections.
These benchmarks aim to evaluate a model's ability to understand, interpret, and reason about code—not just generate it. They test deeper comprehension skills such as code analysis, debugging, optimization, and adherence to coding standards, providing a more holistic assessment of a model's programming capabilities. By emphasizing code reasoning, we can gain deeper insights into where models excel or struggle, ultimately guiding the development of more robust and versatile code language models.

## 4. Code Reasoning Benchmarks

To address the limitations of traditional code generation benchmarks, researchers have developed code reasoning benchmarks that focus on evaluating a model's ability to understand, reason about, and execute code—not just generate it from descriptions. These benchmarks aim to assess deeper comprehension skills such as code interpretation, execution behavior, debugging, optimization, and adherence to best practices, providing a more holistic evaluation of a model's programming capabilities. Several notable code reasoning benchmarks have been introduced.

### *4.1. CRUXEval*

#### 4.1.1. Overview

CRUXEval (Code Reasoning, Understanding, and eXecution Evaluation) [29] is a benchmark specifically designed to examine the abilities of code LMs to reason about the execution behavior of simple Python programs. It addresses limitations in problem variety and testing of execution capabilities by focusing on tasks that require understanding and simulating code execution. The benchmark comprises 800 Python functions ranging from 3 to 13 lines of code, each accompanied by an input-output pair.[29]

 doi:10.20944/preprints202411.1147.v1



#### 4.1.2. Development Approach

The development of CRUXEval [29] involved a three-step process: large-scale distillation, filtering, and data size selection via statistical noise analysis. The researchers used Code Llama 34B to generate candidate functions and inputs, prompting the model with the names of functions from the Python standard library and requesting the generation of functions utilizing those library functions along with five test inputs. This process resulted in the generation of a total of 102,000 functions and 489,306 input-output pairs. The team filtered these generated samples based on code complexity, ensuring that the functions were simple enough to be solvable by a university-level CS graduate without extensive memory requirements.[29]

#### 4.1.3. Purpose

CRUXEval[29] serves as a valuable tool for evaluating LLMs' abilities in two primary tasks:

- **Output Prediction (CRUXEval-O)**: Measures code execution by requiring models to predict the output given a function and an input.
- **Input Prediction (CRUXEval-I)**: Assesses code reasoning and understanding by requiring models to infer the input that would produce a given output when passed to a function.

The development of CRUXEval [29] is motivated by the lack of benchmarks that go beyond code generation and delve into other fundamental aspects of LLMs' code understanding and execution capabilities. The creators of CRUXEval believe that the benchmark provides a crucial means for gaining deeper insights into the strengths and weaknesses of LLMs in handling code-related tasks. [29]

#### 4.1.4. Limitation

While CRUXEval focuses on execution behavior and addresses problem variety, a potential limitation would be:

- Limited Programming Language Support: CRUXEval currently focuses exclusively on Python. This limitation restricts its applicability in assessing LLMs' performance in other programming languages. It is worth noting that CRUXEval-X, a multilingual benchmark containing 19 programming languages, including C++, Rust, and Java, was created as an extension of CRUXEval.
- Scope of Tasks: CRUXEval primarily concentrates on input and output prediction tasks. While these tasks provide a good starting point for evaluating code reasoning and execution, they may not fully capture all aspects of LLM capabilities in understanding and manipulating code.
- Simplicity of Code: CRUXEval focuses on simple Python functions. While this is beneficial for isolating core code reasoning abilities, it might not fully reflect the complexity of real-world code and the challenges LLMs might encounter in handling more intricate programming scenarios.

### *4.2. CodeMind*

#### 4.2.1. Overview

CodeMind [35] is a framework designed to gauge the code reasoning abilities of LLMs through several inductive reasoning tasks. It supports three tasks:

- **Independent Execution Reasoning (IER)**: Evaluates if LLMs can predict the execution output of arbitrary code.
- **Dependent Execution Reasoning (DER)**: Assesses if LLMs can predict the execution output of code they can correctly generate.
- **Specification Reasoning (SR)**: Tests the extent to which LLMs can reason about and implement the specified behavior, covering both code synthesis (generating code from natural language specifications) and code translation (generating code from code in a different programming language).

 doi:10.20944/preprints202411.1147.v1



#### 4.2.2. Development Approach

CodeMind [35] defines a clear distinction between program specification and code implementation. Program specification outlines the intended logic of the program, defining a function S that maps inputs to outputs. Code implementation, represented by function C, is the actual code written based on the specification. A program is considered correct if the code implementation accurately reflects the program specification. CodeMind [35] evaluates LLMs by presenting them with code snippets and corresponding specifications, prompting them to perform the IER, DER, and SR tasks. It then analyzes the LLM's responses to determine their level of code reasoning ability. The framework supports two common code generation tasks:

- **Code synthesis**: Generating code from a natural language specification.
- **Code translation**: Generating code in a different programming language from code provided in another language.

#### 4.2.3. Purpose

CodeMind [35] focuses on the reasoning capabilities of LLMs beyond code generation, particularly their ability to simulate code execution and understand specifications. Its key objectives include:

- Shifting Focus from Code Generation to Code Reasoning
- Identifying Limitations in LLM Reasoning
- Guiding the Development of More Capable LLMs

#### 4.2.4. Limitation

We can acknowledge several limitations associated with CodeMind [35]:

- **Limited Scope of Reasoning Tasks**: Expanding the range of reasoning tasks would provide a more comprehensive evaluation.
- **Focus on Specific Programming Languages**: CodeMind currently supports Java and Python, and extending the framework to include other programming languages would enhance its ability to generalization.
- **Dependence on Benchmark Datasets**: CodeMind's evaluations are based on existing benchmark datasets, which may contain inherent biases or limitations in terms of code complexity, style, or domain coverage.
- **Limited Model Size in Evaluation**: Due to resource constraints, the initial study focused on evaluating LLMs with less than 20B parameters.

### *4.3. Chain of Simulation (CoSm)*

#### 4.3.1. Overview

CoSm [36] is a novel prompting method introduced by [36]. This technique aims to enhance the code simulation capabilities of LLMs by instructing them to simulate code execution line by line, mirroring the analytical process of a compiler. CoSm is designed as an "off-the-shelf" prompting technique, meaning it can be readily integrated with other existing prompting methods such as Chain of Thought (CoT) and Self-Consistency (SC) This benchmark studies the code simulation capabilities of LLMs through six coding challenges that range from simple to complex algorithmic reasoning tasks:

- **Straight-Line Code Simulation**: Tests the ability to simulate simple, sequential instructions consistently.
- **Smart Execution**: Involves programs where the portion relevant to producing the correct output is encapsulated in a shorter, straight-line program.
- **Approximate Computation**: Assesses the capacity to perform multiple independent sub-programs.
- **Redundant Algorithms**: Presents multiple equivalent programs expected to return the same result.

 doi:10.20944/preprints202411.1147.v1



- **Nested Loops**: Connects computational complexity with simulation capabilities.
- **Sorting Algorithms**: Evaluates performance on sorting algorithms of varying complexity, both recursive and iterative.

#### 4.3.2. Development Approach

CoSm [36] was developed in response to observations that while LLMs demonstrated promising code simulation abilities, their success often relied heavily on pattern recognition and memorization rather than a true understanding of code execution. To address this, the researchers designed CoSm [36] to guide LLMs towards simulating each instruction sequentially while maintaining a record of the program trace. The prompting template for CoSm includes instructions for the LLM to:

- Simulate the provided program instruction by instruction.
- Report the trace of the program at the end of each iteration.
- Think step by step and provide the output of the function for a given input

#### 4.3.3. Purpose

The primary purpose of CoSm [36] is to improve the accuracy and reliability of LLMs in simulating code execution. By promoting a step-by-step simulation process, CoSm aims to reduce the LLM's dependence on:

- **Memorization**: CoSm encourages the LLM to focus on the execution logic rather than relying on memorized code patterns
- **Shallow pattern recognition**: CoSm promotes a deeper understanding of the code's semantics, going beyond simple pattern matching

The ultimate goal of CoSm [36] is to bridge the gap between LLMs and their practical application in reasoning, planning, and problem-solving tasks that involve simulating computational processes

#### 4.3.4. Limitation

Although CoSm shows promising results in improving LLM performance for code simulation, it has some limitations:

- **Varied effectiveness across LLMs**: The study revealed that CoSm's impact on performance differed across LLMs. For instance, CoSm consistently enhanced GPT-4's performance on straight-line, critical paths, and nested loop benchmarks but degraded its performance on redundant instruction benchmarks. In contrast, CoSm did not notably improve LLaMA3-70B's performance on straight-line and critical path benchmarks but led to a marginal improvement on nested loop benchmarks. This suggests that the effectiveness of CoSm might be influenced by the specific architecture and training data of different LLMs.
- **Limited Scope**: The current research on CoSm primarily focuses on code simulation tasks. Further investigation is needed to explore its applicability and potential benefits in other algorithmic reasoning tasks.

### *4.4. CRQBench*

#### 4.4.1. Overview

CRQBench [37] is a benchmark designed to assess the code-reasoning abilities of LLMs. It represents a shift toward evaluating code reasoning in real-world programming scenarios. It focuses on evaluating how well LLMs can understand and answer questions related to C++ code within a specific context. The benchmark comprises 100 C++ code-reasoning questions with answers derived from contextualized code review comments found in the CodeReviewer dataset, a collection of code review data from GitHub [37]. The CRQBench [37] focuses specifically on two types of code-reasoning queries:

 doi:10.20944/preprints202411.1147.v1



- **VALUE Queries**: These queries focus on understanding the value of a specific program element (e.g., variable, expression) within the given code context.
- **EQUIV Queries**: These queries center on determining the equivalence of two program elements within the provided context.

It has 2 notable features:

- **Authentic Semantic Queries**: The questions are non-synthetic and tied to real code contexts, requiring deep semantic understanding.
- **Includes Code Context and Answers**: Each question is accompanied by the relevant code context and a reference answer.

#### 4.4.2. Development Approach

The creation of CRQBench [37] involved a multi-stage process using an LLM assistant and human validation:

1. **Data Source**: The initial data source is the CodeReviewer dataset, which provides a large corpus of code review comments.
2. **Code Reasoning Classification**: An LLM is used to classify code review comments and identify those that contain code-reasoning questions
3. **Rephrasing Comments as CRQs**: The selected code-reasoning comments are then rephrased into concise questions grounded over specific program elements.
4. **Query Type Classification**: Each rephrased question is further classified as either a VALUE or EQUIV query.
5. **Human Validation**: All stages involve human inspection to ensure the quality and relevance of the generated questions.

There are two main points in development approach of CRQBench

- **Cooperative LLM and Human-in-the-Loop Methodology**: Utilizes in-context learning to filter and rephrase code reasoning questions from code review comments, reducing human curator effort.
- **Focus on Semantic Depth**: Excludes superficial style and refactoring questions to concentrate on meaningful semantic reasoning.

#### 4.4.3. Purpose

CRQBench [37] aims to evaluate the semantic reasoning abilities of models in authentic programming situations, addressing limitations related to real-world problem representation and limited coverage of code understanding. Key objectives include:

- **Evaluating Semantic Reasoning in Isolation**: CRQBench focuses specifically on code reasoning, isolating this ability from other aspects of coding like generation or debugging.
- **Using Real-World, Contextualized Questions**: The questions in CRQBench are derived from actual code review comments, providing a more realistic and relevant evaluation of how LLMs might be used in practice
- **Reducing Manual Curation Effort**: The LLM-assisted approach significantly reduces the human effort required to create a high-quality code-reasoning benchmark.

#### 4.4.4. Limitation

We can pinpoint potential limitations include:

- **Focus on a Single Programming Language**: CRQBench [37] currently focuses solely on C++ code. Extending the benchmark to include other programming languages would be beneficial for a more comprehensive evaluation of LLM capabilities.

 doi:10.20944/preprints202411.1147.v1



- **Limited Number of Questions**: With only 100 questions, CRQBench may not fully capture the diversity and complexity of real-world code reasoning scenarios. Increasing the number and diversity of questions would enhance the benchmark's representativeness.
- **Dependence on the Quality of the CodeReviewer Dataset**: The quality and representativeness of the questions in CRQBench are inherently tied to the quality and diversity of code review comments in the CodeReviewer dataset. Any biases or limitations present in the dataset would propagate to the benchmark.
- **Potential for LLM Bias in Curation**: Using an LLM assistant in the curation process may introduce biases inherent to the LLM itself. This could result in a benchmark that unintentionally favors certain types of questions or reasoning patterns.

### *4.5. SpecEval*

#### 4.5.1. Overview

SpecEval [38] is a novel black-box evaluation framework designed to assess the code comprehension abilities of Large Language Models (LLMs) using formal program specifications as a representation of program semantics. It aims to evaluate the ability of LLMs to understand code beyond simple input-output relationships, focusing on their capacity to articulate program behavior using formal specifications. SpecEval [38] consists of four core tasks related to program specifications, each progressively more challenging

- **Specification Correctness Judgment**: Determining whether a given specification accurately describes the behavior of a target program.
- **Specification Candidates Selection**: Choosing the most accurate and comprehensive specification from a set of candidates for a given program.
- **Specification Infilling**: Completing partially provided specifications by filling in missing components while maintaining consistency with the program's behavior.
- **Specification Generation**: Generating a complete and accurate specification for a given program from scratch.

#### 4.5.2. Development Approach

SpecEval [38] employs a black-box evaluation approach, meaning it does not require access to the internal workings of the LLMs being evaluated. The framework utilizes existing datasets of 204 Java programs with manually crafted, verifiable ground-truth specifications written in the Java Modeling Language (JML) style. SpecEval [38] assesses LLMs by presenting them with these programs and their specifications, asking them to perform the four core tasks. The accuracy and completeness of the LLM-generated outputs are then evaluated against the ground-truth specifications using automated verification tools. To further assess the robustness and consistency of LLM understanding, SpecEval incorporates two key analytical techniques:

- **Counterfactual Analysis**: This involves introducing controlled, semantics-preserving perturbations to the input code, such as swapping variable names, altering operators, or flipping conditional statements. By comparing the LLM's performance on the original code and its perturbed variants, SpecEval aims to evaluate the LLM's sensitivity to these changes and its ability to maintain consistent understanding despite variations in code expression.
- **Progressive Consistency Analysis**: This technique analyzes the consistency of an LLM's performance across the four core tasks, which are designed with sequential dependencies.

#### 4.5.3. Purpose

SpecEval [38] addresses the limitation of emphasizing code generation over understanding by focusing on the semantic comprehension of programs. It evaluates how well models learn program semantics and can reason about code behavior based on formal specifications.

 doi:10.20944/preprints202411.1147.v1



#### 4.5.4. Limitation

We can identify several potential limitations of SpecEval [38]:

- Reliance on Specific Formal Specification Language
- Potential Bias in Benchmark Datasets
- Limited Scope of Code Complexity
- Dependence on the Quality of Specification Verifiers
- Focus on Java limits its applicability to assessing models' capabilities in other programming languages.

### *4.6. REval*

#### 4.6.1. Overview

REval [39] is a framework designed to comprehensively evaluate the code reasoning capabilities of LLMs, particularly their ability to predict code execution behaviors without actually executing the code. It focuses on both runtime behavior reasoning, which involves predicting intermediate states during code execution, and incremental consistency evaluation, which assesses the consistency of an LLM's reasoning across sequentially related tasks of increasing difficulty.

- **Runtime Behavior Reasoning**: Tests the model's ability to predict the runtime behavior of code.
- **Incremental Consistency Evaluation**: Measures the consistency of the model's reasoning as more information is provided.
- **Code Execution Behavior**: Evaluates how well the model aligns with actual code execution.
- **Pre/Post-Execution Information**: Assesses the model's ability to reason with or without execution traces.
- **Runtime Information**: Incorporates runtime details to test the model's understanding of dynamic code behavior.

#### 4.6.2. Development Approach

REval [39] utilizes existing code benchmarks like HumanEval [26] and ClassEval [40], adapting them to include tasks that assess runtime behavior and consistency. The REval framework is constructed based on two key evaluation components:

- **Runtime Behavior Reasoning**: This component focuses on evaluating an LLM's ability to predict the dynamic characteristics of a program during execution. REval introduces four specific tasks to assess this ability
  - Code Coverage Prediction (CCP)
  - Program State Prediction (PSP)
  - Execution Path Prediction (EPP)
  - Output Prediction (OP)
- **Incremental Consistency Evaluation**: This component focuses on evaluating the logical consistency of an LLM's reasoning process across a series of related tasks with increasing complexity. It assesses whether the LLM's predictions for later tasks are consistent with its predictions for earlier tasks, reflecting a more human-like, step-by-step reasoning approach.
  REval [39] introduces the Incremental Consistency (IC) score, which quantifies the degree to which an LLM maintains consistency in its predictions across the four runtime behavior reasoning tasks.

#### 4.6.3. Purpose

REval addresses the limitation of minimal testing of execution capabilities by directly evaluating models on their ability to reason about runtime behavior, execution consistency, and the impact of code execution on program state.

 doi:10.20944/preprints202411.1147.v1



- Comprehensive Evaluation of Code Reasoning
- Assessing Reasoning Consistency
- Highlighting Areas for Improvement

#### 4.6.4. Limitation

we can point out several limitations associated with REval [39]:

- **Reliance on Existing Benchmarks**: While REval adapts existing benchmarks, it still inherits any inherent limitations or biases present in the original datasets.
- **Limited Scope of Code Complexity**: REval focuses on evaluating runtime behavior in relatively simple programs, and it's unclear how well these evaluations generalize to more complex code structures found in real-world software development.
- **Evaluation of Specific Models**: The initial evaluation of REval was conducted on a limited set of LLMs, and further research is needed to assess its effectiveness and generalizability across a wider range of models.

### *4.7. LiveCodeBench*

#### 4.7.1. Overview

LiveCodeBench [41] is a comprehensive and contamination-free benchmark designed to evaluate the code capabilities of Large Language Models (LLMs). It consists of over 500 coding problems collected from contests on LeetCode, AtCoder, and CodeForces between May 2023 and May 2024. LiveCodeBench [41] aims to provide a holistic evaluation of LLM code capabilities by including tasks like self-repair, code execution, and test output prediction, in addition to code generation. It also utilizes problems with robust test cases and balanced difficulty levels. The benchmark emphasizes avoiding data contamination by continuously updating with new problems, ensuring that the models are evaluated on unseen data

#### 4.7.2. Development Approach

LiveCodeBench [41] is developed based on three main principles: live updates to prevent contamination, holistic evaluation, and high-quality problems and tests

- **Live Updates**: Problems are collected from recent coding contests to ensure they are not included in the training data of LLMs. This helps avoid contamination, a significant issue in static benchmarks like HumanEval [26] and MBPP [27]
- **Holistic Evaluation**: LiveCodeBench assesses LLMs across a broader range of coding-related capabilities than just code generation. The benchmark includes scenarios for self-repair, code execution, and test output prediction, capturing a more comprehensive picture of LLMs' programming skills
- **High-Quality Problems and Tests**: LiveCodeBench [41] uses problems from reputable competition websites, ensuring their quality and clarity. For each problem, multiple test cases are provided, averaging around 17 per problem, which helps in robust and meaningful evaluations

#### 4.7.3. Purpose

LiveCodeBench aims to offer a more comprehensive and reliable evaluation of code LLMs compared to existing benchmarks. It tries to address limitations related to data leakage, problem variety, and overemphasis on code generation by providing a dynamic and comprehensive evaluation platform that simulates real-world coding challenges.

- **Contamination-Free Evaluation**: The live update mechanism addresses the issue of data contamination by using problems from recent contests that LLMs are unlikely to have been trained on

 doi:10.20944/preprints202411.1147.v1



- **Holistic Assessment of Code Capabilities**: The inclusion of scenarios beyond code generation, like self-repair, code execution, and test output prediction, provides a more comprehensive understanding of LLM code capabilities.
- **Understanding LLM Strengths and Weaknesses**: LiveCodeBench's evaluation across various coding scenarios allows for a deeper analysis of the strengths and weaknesses of different LLMs, providing valuable insights for model development and improvement.

#### 4.7.4. Limitation

We can identify the following limitations for LiveCodeBench [41]:

- **Benchmark Size**: The number of problems used in the evaluation, especially for scenarios beyond code generation, might be relatively small, potentially leading to some noise in the performance measurements. The limited size is primarily due to the focus on using recent problems to avoid contamination [41].
- **Focus on Python**: Currently, LiveCodeBench only includes Python problems, which limits its ability to assess LLM capabilities in other programming languages. Expanding the benchmark to include other languages is feasible, but requires the development of appropriate evaluation engines.
- **Robustness to Prompts**: The performance of LLMs can be significantly influenced by the prompt design. LiveCodeBench [41] either does not tune prompts across models or makes only minor adjustments, which could introduce performance variations in the results. More robust prompt engineering or a standardized prompt format could mitigate this issue.
- **Problem Domain**: The problems in LiveCodeBench [41] are sourced from coding competitions, which may not fully represent the diversity of coding tasks in real-world software development. It's important to recognize that LLMs may perform differently when faced with open-ended, unconstrained problems commonly encountered in practical settings.
- **Potentially Noisy Evaluation Due to Small Problem Set**: This limitation was observed when evaluating models on problems released after a specific cutoff date to avoid contamination. The use of smaller problem sets might introduce variability in performance estimates.

### *4.8. CodeMMLU*

#### 4.8.1. Overview

CodeMMLU [42] is a multi-task, multiple-choice question-answering (MCQA) benchmark specifically crafted to evaluate the depth of software and code understanding exhibited by Large Language Models (LLMs). The benchmark comprises over 10,000 questions spanning a variety of domains, including code analysis, defect detection, software engineering principles, and encompassing multiple programming languages. The questions are structured in a multiple-choice format, requiring LLMs to select the most appropriate answer from a set of options. also based on multi-task feature it assesses various aspects of code understanding, not just code generation.

#### 4.8.2. Development Approach

CodeMMLU [42] employs a multi-stage process to curate its extensive question set:

- **Source Selection**: CodeMMLU draws questions from a diverse range of sources, including established benchmarks like HumanEval [26], QuixBugs [43], and IBM CodeNet [44], as well as real-world programming challenges curated from platforms like LeetCode, GeeksForGeeks, and W3Schools
- **Task Construction**: The benchmark encompasses five distinct MCQA programming tasks: Code Completion, Code Repair, Defect Detection, Fill in the blank, and Others. These tasks are designed to assess a model's proficiency in core programming capabilities like composition, comprehension, debugging, and modification

 doi:10.20944/preprints202411.1147.v1



- **Distractor Generation**: For each question, LLMs are used to generate plausible but incorrect answer choices, referred to as distractors. These distractors are carefully crafted to be contextually relevant and challenge the model's understanding of the underlying code and concepts. The correctness of both the correct answers and the generated distractors is verified through execution to ensure accuracy and difficulty
- **Normalization and Filtering**: To eliminate bias and focus on code logic, variable and function names within the code snippets are normalized, replacing specific identifiers with generic placeholders. Additionally, rigorous filtering mechanisms are applied to eliminate duplicate questions, ensure appropriate difficulty levels, and maintain diversity across tasks, domains, and languages

#### 4.8.3. Purpose

CodeMMLU [42] aims to go beyond the traditional focus on code generation in LLM evaluation and provide a more in-depth assessment of their ability to actually understand and reason about code. This includes:

- **Assessing Code Comprehension**: By presenting LLMs with a diverse set of challenging questions in an MCQA format, CodeMMLU probes their ability to comprehend code semantics, identify errors, and apply software engineering principles. This goes beyond surface-level understanding and focuses on the model's capacity to grasp the underlying logic and functionality of code [42].
- **Identifying Bias and Limitations**: The benchmark's comprehensive coverage across tasks, domains, and programming languages allows researchers to uncover potential biases and limitations in LLMs, particularly those related to specific languages, coding styles, or problem-solving approaches
- **Advancing AI-Assisted Software Development**: By rigorously evaluating LLM code understanding capabilities, CodeMMLU [42] aims to drive the development of more reliable and capable coding assistants. Identifying and addressing the shortcomings revealed by the benchmark is crucial for creating LLMs that can effectively support human programmers in complex software development tasks

#### 4.8.4. Limitation

We can highlight the following potential limitations of CodeMMLU [42]:

- **Limited Scope of Creative Code Generation**:The MCQA format, while effective for evaluating code comprehension, may not fully capture the creative aspects of code generation or a model's ability to generate optimized or elegant code solutions [42].
- **Potential Bias from Source Material**:Despite efforts to diversify sources, inherent biases or limitations in the original datasets used to construct CodeMMLU [42] could potentially influence the benchmark's representativeness or introduce unintended biases in the evaluation process.
- **Ongoing Development and Expansion**: As an evolving benchmark, CodeMMLU [42] acknowledges the need for ongoing expansion and refinement. This includes adding more complex tasks, improving the balance between theoretical and real-world scenarios, and incorporating user feedback to enhance the benchmark's robustness and relevance.

### *4.9. XCODEEVAL*

#### 4.9.1. Overview

XCODEEVAL [45] is an Execution-Based, large-scale multilingual multitask benchmark specifically designed for evaluating code-based Large Language Models (LLMs). It comprises 25 million document-level coding examples, totaling 16.5 billion tokens, collected from around 7,500 unique algorithmic problems sourced from codeforces.com. It covers 17 programming languages and includes seven distinct tasks: two classification, three generative, and two retrieval. XCODEEVAL utilizes an

 doi:10.20944/preprints202411.1147.v1



execution-based evaluation protocol, meaning it assesses the correctness of generated code by running it against a comprehensive set of unit tests. It includes seven tasks involving code understanding, generation, translation, and retrieval [45].

#### 4.9.2. Development Approach

The creation of XCODEEVAL [45] involved several key steps:

- **Data Collection**:The dataset was collected from codeforces.com, a platform that hosts coding competitions. Each sample in XCODEEVAL includes a problem description, a code solution, and metadata such as difficulty level, language, and problem tags [45].
- **Data Splitting**: To prevent data leakage, a held-out set of 1,354 problems was created for validation and testing. The validation and test splits were then created while maintaining a balanced tag distribution across the splits and ensuring that all tags in the evaluation sets were also present in the training data.
- **Data Selection**: A balanced data selection schema was implemented to ensure representation across various problem attributes, such as programming language, problem tags, and difficulty level.
- **Unit Test Creation**: A large number of unit tests (averaging 50 per problem) were developed to support execution-based evaluation for the relevant tasks.
- **ExecEval Development**: A multilingual, distributed, and secure execution engine called ExecEval [45] was developed to support the execution-based evaluation. ExecEval can handle 44 different compiler/interpreter versions across 11 programming languages and provides six possible execution outcomes, including compilation errors, runtime errors, and exceeding memory or time limits.

#### 4.9.3. Purpose

XCODEEVAL [45] aims to address the limitations of existing code-related benchmarks and drive advancements in more general-purpose problem-solving LLMs. Its key purposes include:

- **Comprehensive Evaluation**: By encompassing multiple tasks, languages, and an execution-based evaluation protocol, XCODEEVAL [45] offers a more holistic and rigorous assessment of LLMs' code-related abilities compared to previous benchmarks, which often focused on single tasks or relied on lexical overlap metrics
- **Promoting Reasoning Abilities**: The benchmark's focus on problem-solving tasks requires LLMs to go beyond simple code generation and demonstrate understanding of complex problem descriptions, algorithmic concepts, and the ability to generate functionally correct code
- **Assessing Multilinguality**: The inclusion of 17 programming languages allows researchers to evaluate the cross-lingual capabilities of LLMs and analyze the parallelism between different programming languages
- **Supporting Global Understanding**: XCODEEVAL's execution-based evaluation at the document level encourages models to develop a global understanding of the code, considering interactions between different code segments and their overall functionality [45].

#### 4.9.4. Limitation

There are several limitations in XCODEEVAL [45]:

- **Domain Diversity**: The data is sourced from a single platform (codeforces.com), which could limit the diversity of coding styles and problem domains represented in the benchmark
- **Resource Discrepancies**: There are significant differences in the number of samples available for different programming languages, which could introduce biases in the evaluation
- **Lack of Modularity**: Most of the code in XCODEEVAL is at the document level and often written in a non-modular way, which may not reflect real-world software development practices.

 doi:10.20944/preprints202411.1147.v1



- **Lack of real-world coding challenges**: While XCODEEVAL is extensive and multilingual, it might not cover the full spectrum of real-world coding challenges, such as those requiring complex interactions with external systems or handling edge cases not explicitly addressed in the problem sets.

### *4.10. CodeJudge-Eval*

#### 4.10.1. Overview

CodeJudge-Eval (CJ-Eval) [46] is a benchmark designed to assess the code understanding abilities of LLMs by challenging them to judge the correctness of provided code solutions rather than generating code themselves. This benchmark utilizes problems sourced from the APPS (Automated Programming Progress Standard) [21] test set, which includes a diverse set of 5,000 coding problems across three difficulty levels: introductory, interview, and competition. CJ-Eval [46] uses a fine-grained judging system that goes beyond simple pass/fail criteria, taking into account various error types and compilation issues to assess code correctness more comprehensively. The candidate solutions provided for each problem are generated using 16 different LLMs, ensuring a diverse set of potential solutions for the models to judge

#### 4.10.2. Development Approach

The pipeline for CJ-Eval [46] describes as a multi-step process:

- **Problem Selection**: Problems are sourced from the APPS [21] test set, providing a diverse and challenging collection of coding tasks.
- Candidate Code Generation: Sixteen different LLMs generate a pool of candidate solutions for each problem, encompassing both correct and incorrect implementations.
- **Fine-grained Verdict Construction**: A local judging system with a comprehensive set of test cases determines the correctness of each candidate solution, providing detailed verdicts beyond simple binary classifications.
- **Data Filtering**: The initial set of 80,000 solutions is filtered down to 1,860 solutions to create a curated benchmark, removing redundant or low-quality solutions.
- **Multiple-Choice Format**: The filtered solutions are structured into multiple-choice questions for LLMs to evaluate, requiring them to select the correct verdict from a set of options.

#### 4.10.3. Purpose

CJ-Eval [46] aims to provide a more robust evaluation of code understanding in LLMs compared to traditional language-to-code benchmarks, which primarily focus on code generation and can be susceptible to memorization or data leakage issues. By positioning LLMs as code judges and requiring them to assess the correctness of various solutions, CJ-Eval encourages deeper code comprehension and reasoning. This approach aligns with the idea that being able to judge the correctness of solutions demonstrates a higher level of understanding than merely being able to generate a solution [46]. The benchmark's design emphasizes the importance of evaluating code understanding as a separate skill from code generation, highlighting the need for more comprehensive assessments of LLM capabilities in the coding domain

#### 4.10.4. Limitation

While CJ-Eval [46] offers a novel perspective for assessing code comprehension, the source, It's possible to highlight the following potential limitations:

- **Complementary Evaluation**: CJ-Eval does not evaluate code generation, so it should be used in conjunction with language-to-code evaluations for a complete assessment of LLM code understanding.

 doi:10.20944/preprints202411.1147.v1



- **Bias in Problem Selection**: Relying on problems from a single source like the APPS [21] test set might introduce biases or limitations in the types of coding tasks represented.
- **Potential for Adversarial Examples**: Similar to other LLM-as-a-judge approaches, CJ-Eval could be vulnerable to adversarial examples or carefully crafted solutions designed to mislead the evaluating models.

It's also worth noting that the authors mention a potential issue with model responses not following the specified format during evaluation. This highlights the need for clear instructions and format constraints to ensure consistent and interpretable evaluations.

### *4.11. CRUXEval-X*

#### 4.11.1. Overview

CRUXEval-X [47] is a Benchmark for Multilingual Code Reasoning, Understanding, and Execution. It extends the Python-based CRUXEval benchmark to 19 programming languages, comprising at least 600 samples for each language and totaling 12,660 subjects and 19,000 test cases for evaluating input/output reasoning capabilities.

#### 4.11.2. Development Approach

CRUXEval-X [47] employs a fully automated, test-guided approach for its construction. It Uses an iterative generation-and-repair process based on execution feedback. The pipeline involves:

1. **Translation of Function Signatures**: Translates function signatures via mapping variable type annotations.
2. **Translation of Test Cases**: Employs a rule-based approach to translate Python test cases into other programming languages.
3. **Integration of Multiple LLMs**: Integrating multiple LLMs for code translation, utilizing an iterative generate-and-repair process guided by execution feedback (e.g., compilation and runtime errors)

#### 4.11.3. Purpose

CRUXEval-X [47] aims to address the language bias prevalent in existing code benchmarks that heavily favor Python. By evaluating LLMs across a wide range of programming languages, it provides a more comprehensive assessment of their code reasoning capabilities and cross-language generalization abilities

#### 4.11.4. Limitation

While CRUXEval-X [47] extends language coverage, it might not capture the subtle differences and nuances in programming paradigms and idioms across various languages. For instance, the dynamic typing of Python versus the static typing of C++ requires careful consideration during translation and evaluation. Additionally, the automated translation process, while efficient, could potentially introduce inconsistencies or inaccuracies that may affect the benchmark's reliability.

## 5. Future Directions for Code Reasoning Benchmarks

The rapid evolution of LLMs' code capabilities necessitates more sophisticated evaluation frameworks. Based on our analysis, we identify several critical directions for the development of future code reasoning benchmarks:

### *5.1. Comprehensive Skill Assessment*

Future benchmarks should expand beyond basic input/output prediction to evaluate a broader spectrum of programming capabilities:

 doi:10.20944/preprints202411.1147.v1



- **Runtime Behavior Analysis**: Assessing models' ability to predict and explain code execution paths, variable states, and control flow
- **Debugging and Maintenance**: Evaluating capabilities in error identification, code repair, and optimization
- **Code Comprehension**: Testing deeper understanding through code summarization and documentation generation
- **Program Transformation**: Measuring ability to refactor, optimize, and modernize existing code

### *5.2. Enhanced Evaluation Methodologies*

To provide more rigorous assessment, future benchmarks should incorporate:

- **Formal Program Specifications**: Using precise semantic definitions to evaluate understanding beyond input-output behavior
- **Multi-Stage Evaluation**: Assessing incremental reasoning abilities through progressively complex tasks
- **Qualitative Analysis**: Including metrics for code readability, maintainability, and adherence to best practices
- **Context-Aware Testing**: Evaluating performance within larger code bases and system architectures

### *5.3. Robustness and Reliability*

Several measures can enhance the reliability of benchmark evaluations:

- **Contamination Prevention**: Incorporating continually updated problems from recent programming competitions
- **Cross-Platform Validation**: Testing across different development environments and frameworks
- **Edge Case Coverage**: Including comprehensive test suites that cover boundary conditions and error scenarios
- **Consistency Metrics**: Measuring the reliability and reproducibility of model responses

### *5.4. Language and Domain Coverage*

Future benchmarks should expand their scope to include:

- **Diverse Programming Languages**: Supporting evaluation across multiple programming paradigms and languages
- **Domain-Specific Scenarios**: Including specialized programming tasks from various technical domains
- **Framework-Specific Testing**: Evaluating understanding of popular programming frameworks and libraries
- **Cross-Language Translation**: Assessing ability to transfer knowledge between programming languages

### *5.5. Advanced Evaluation Techniques*

Innovation in evaluation methodologies should explore:

- **Interactive Assessment**: Implementing dynamic evaluation scenarios that adapt to model responses
- **Collaborative Problem-Solving**: Evaluating models' ability to work with human developers
- **Advanced Prompting Strategies**: Systematically analyzing the impact of different prompting techniques
- **Multi-Model Comparison**: Facilitating standardized comparison across different model architectures

 doi:10.20944/preprints202411.1147.v1



### *5.6. Community and Infrastructure*

The development of future benchmarks should prioritize:

- **Open Collaboration**: Fostering community involvement in benchmark development and maintenance
- **Standardized Tools**: Creating robust evaluation infrastructure and tooling
- **Reproducible Results**: Ensuring transparency and reproducibility of benchmark results
- **Regular Updates**: Maintaining relevance through continuous evolution of evaluation criteria

This evolution in benchmark development is crucial for accurately assessing and driving improvements in LLMs' code reasoning capabilities. By implementing these recommendations, future benchmarks can provide more comprehensive and nuanced evaluations, ultimately contributing to the development of more capable and reliable code-aware AI systems.

## 6. Conclusions

The rapid development of Large Language Models (LLMs) in software engineering and code understanding presents a critical need for sophisticated benchmarks to evaluate their code reasoning capabilities. While traditional code generation benchmarks have paved the way for assessing baseline functionality, they fall short of capturing the deeper reasoning, comprehension, and multi-step problem-solving required in real-world programming. This review underscores the necessity for benchmarks that go beyond syntactic correctness and functional outcomes to encompass runtime behavior, semantic comprehension, and cross-language reasoning. Emerging evaluation frameworks, such as CRUXEval [29], CodeMind [35], CoSm [36], and REval [39], illustrate significant progress toward assessing LLMs' deeper understanding of programming logic. These benchmarks offer valuable insights into how models interpret, simulate, and reason about code behavior, addressing key limitations of traditional metrics that focus mainly on code generation. However, they also highlight critical gaps, such as limited language coverage, lack of real-world coding scenarios, and insufficient problem variety. To address these challenges, future benchmarks should incorporate contamination-resistant evaluation methods, expand language and domain coverage, and adopt formal program specifications for a more comprehensive skill assessment. In moving forward, the focus should be on developing benchmarks that not only evaluate an LLM's ability to generate correct code but also measure its robustness in reasoning, adaptability to diverse languages, and effectiveness in complex scenarios. This progression in benchmark sophistication will provide researchers and developers with the tools necessary to build and refine LLMs capable of supporting the dynamic demands of AI-assisted software development. By continually advancing evaluation frameworks, the AI community can work towards creating LLMs with truly enhanced code reasoning capabilities that meet the practical needs of the programming industry.

## References

1. Chen, L.; Guo, Q.; Jia, H.; Zeng, Z.; Wang, X.; Xu, Y.; Wu, J.; Wang, Y.; Gao, Q.; Wang, J.; Ye, W.; Zhang, S. A Survey on Evaluating Large Language Models in Code Generation Tasks, 2024, [arXiv:cs.SE/2408.16498].
2. Guo, Z.; Jin, R.; Liu, C.; Huang, Y.; Shi, D.; Supryadi.; Yu, L.; Liu, Y.; Li, J.; Xiong, B.; Xiong, D. Evaluating Large Language Models: A Comprehensive Survey, 2023, [arXiv:cs.CL/2310.19736].
3. Chang, Y.; Wang, X.; Wang, J.; Wu, Y.; Yang, L.; Zhu, K.; Chen, H.; Yi, X.; Wang, C.; Wang, Y.; Ye, W.; Zhang, Y.; Chang, Y.; Yu, P.S.; Yang, Q.; Xie, X. A Survey on Evaluation of Large Language Models, 2023, [arXiv:cs.CL/2307.03109].
4. Ali, M.; Rao, P.; Mai, Y.; Xie, B. Using Benchmarking Infrastructure to Evaluate LLM Performance on CS Concept Inventories: Challenges, Opportunities, and Critiques. Proceedings of the 2024 ACM Conference on International Computing Education Research - Volume 1; Association for Computing Machinery: New York, NY, USA, 2024; ICER '24, p. 452–468. doi:10.1145/3632620.3671097.
5. Mondorf, P.; Plank, B. Beyond Accuracy: Evaluating the Reasoning Behavior of Large Language Models – A Survey, 2024, [arXiv:cs.CL/2404.01869].

 doi:10.20944/preprints202411.1147.v1




6. Zhang, Z.; Chen, C.; Liu, B.; Liao, C.; Gong, Z.; Yu, H.; Li, J.; Wang, R. Unifying the Perspectives of NLP and Software Engineering: A Survey on Language Models for Code, 2024, [arXiv:cs.CL/2311.07989].
7. Haque, M.M.A.; Ahmad, W.U.; Lourentzou, I.; Brown, C. FixEval: Execution-based Evaluation of Program Fixes for Programming Problems, 2023, [arXiv:cs.SE/2206.07796].
8. Levenshtein, V.I. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics. Doklady* **1965**, *10*, 707–710.
9. Papineni, K.; Roukos, S.; Ward, T.; Zhu, W.J. BLEU: a method for automatic evaluation of machine translation. Proceedings of the 40th Annual Meeting on Association for Computational Linguistics; Association for Computational Linguistics: USA, 2002; ACL '02, p. 311–318. doi:10.3115/1073083.1073135.
10. Lin, C.Y. ROUGE: A Package for Automatic Evaluation of Summaries. Text Summarization Branches Out; Association for Computational Linguistics: Barcelona, Spain, 2004; pp. 74–81.
11. Banerjee, S.; Lavie, A. METEOR: An Automatic Metric for MT Evaluation with Improved Correlation with Human Judgments. Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization; Goldstein, J.; Lavie, A.; Lin, C.Y.; Voss, C., Eds.; Association for Computational Linguistics: Ann Arbor, Michigan, 2005; pp. 65–72.
12. Popović, M. chrF: character n-gram F-score for automatic MT evaluation. Proceedings of the Tenth Workshop on Statistical Machine Translation; Bojar, O.; Chatterjee, R.; Federmann, C.; Haddow, B.; Hokamp, C.; Huck, M.; Logacheva, V.; Pecina, P., Eds.; Association for Computational Linguistics: Lisbon, Portugal, 2015; pp. 392–395. doi:10.18653/v1/W15-3049.
13. Ren, S.; Guo, D.; Lu, S.; Zhou, L.; Liu, S.; Tang, D.; Sundaresan, N.; Zhou, M.; Blanco, A.; Ma, S. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis, 2020, [arXiv:cs.SE/2009.10297].
14. Tran, N.; Tran, H.; Nguyen, S.; Nguyen, H.; Nguyen, T. Does BLEU Score Work for Code Migration? 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC). IEEE, 2019, p. 165–176. doi:10.1109/icpc.2019.00034.
15. Eghbali, A.; Pradel, M. CrystalBLEU: Precisely and Efficiently Measuring the Similarity of Code. Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering; Association for Computing Machinery: New York, NY, USA, 2023; ASE '22. doi:10.1145/3551349.3556903.
16. Zhang, T.; Kishore, V.; Wu, F.; Weinberger, K.Q.; Artzi, Y. BERTScore: Evaluating Text Generation with BERT, 2020, [arXiv:cs.CL/1904.09675].
17. Zhou, S.; Alon, U.; Agarwal, S.; Neubig, G. CodeBERTScore: Evaluating Code Generation with Pretrained Models of Code, 2023, [arXiv:cs.SE/2302.05527].
18. Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; de Oliveira Pinto, H.P.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; Ray, A.; Puri, R.; Krueger, G.; Petrov, M.; Khlaaf, H.; Sastry, G.; Mishkin, P.; Chan, B.; Gray, S.; Ryder, N.; Pavlov, M.; Power, A.; Kaiser, L.; Bavarian, M.; Winter, C.; Tillet, P.; Such, F.P.; Cummings, D.; Plappert, M.; Chantzis, F.; Barnes, E.; Herbert-Voss, A.; Guss, W.H.; Nichol, A.; Paino, A.; Tezak, N.; Tang, J.; Babuschkin, I.; Balaji, S.; Jain, S.; Saunders, W.; Hesse, C.; Carr, A.N.; Leike, J.; Achiam, J.; Misra, V.; Morikawa, E.; Radford, A.; Knight, M.; Brundage, M.; Murati, M.; Mayer, K.; Welinder, P.; McGrew, B.; Amodei, D.; McCandlish, S.; Sutskever, I.; Zaremba, W. Evaluating Large Language Models Trained on Code, 2021, [arXiv:cs.LG/2107.03374].
19. Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Dal Lago, A.; Hubert, T.; Choy, P.; de Masson d'Autume, C.; Babuschkin, I.; Chen, X.; Huang, P.S.; Welbl, J.; Gowal, S.; Cherepanov, A.; Molloy, J.; Mankowitz, D.J.; Sutherland Robson, E.; Kohli, P.; de Freitas, N.; Kavukcuoglu, K.; Vinyals, O. Competition-level code generation with AlphaCode. *Science* **2022**, *378*, 1092–1097. doi:10.1126/science.abq1158.
20. Yu, T.; Zhang, R.; Yang, K.; Yasunaga, M.; Wang, D.; Li, Z.; Ma, J.; Li, I.; Yao, Q.; Roman, S.; Zhang, Z.; Radev, D. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task, 2019, [arXiv:cs.CL/1809.08887].
21. Hendrycks, D.; Basart, S.; Kadavath, S.; Mazeika, M.; Arora, A.; Guo, E.; Burns, C.; Puranik, S.; He, H.; Song, D.; Steinhardt, J. Measuring Coding Challenge Competence With APPS, 2021, [arXiv:cs.SE/2105.09938].
22. Zhou, A.; Yan, K.; Shlapentokh-Rothman, M.; Wang, H.; Wang, Y.X. Language Agent Tree Search Unifies Reasoning Acting and Planning in Language Models, 2024, [arXiv:cs.AI/2310.04406].
23. Allamanis, M.; Panthaplackel, S.; Yin, P. Unsupervised Evaluation of Code LLMs with Round-Trip Correctness, 2024, [arXiv:cs.SE/2402.08699].

24. Dong, Y.; Ding, J.; Jiang, X.; Li, G.; Li, Z.; Jin, Z. CodeScore: Evaluating Code Generation by Learning Code Execution, 2024, [arXiv:cs.SE/2301.09043].
25. Mozannar, H.; Chen, V.; Alsobay, M.; Das, S.; Zhao, S.; Wei, D.; Nagireddy, M.; Sattigeri, P.; Talwalkar, A.; Sontag, D. The RealHumanEval: Evaluating Large Language Models' Abilities to Support Programmers, 2024, [arXiv:cs.SE/2404.02806].
26. Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; de Oliveira Pinto, H.P.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; Ray, A.; Puri, R.; Krueger, G.; Petrov, M.; Khlaaf, H.; Sastry, G.; Mishkin, P.; Chan, B.; Gray, S.; Ryder, N.; Pavlov, M.; Power, A.; Kaiser, L.; Bavarian, M.; Winter, C.; Tillet, P.; Such, F.P.; Cummings, D.; Plappert, M.; Chantzis, F.; Barnes, E.; Herbert-Voss, A.; Guss, W.H.; Nichol, A.; Paino, A.; Tezak, N.; Tang, J.; Babuschkin, I.; Balaji, S.; Jain, S.; Saunders, W.; Hesse, C.; Carr, A.N.; Leike, J.; Achiam, J.; Misra, V.; Morikawa, E.; Radford, A.; Knight, M.; Brundage, M.; Murati, M.; Mayer, K.; Welinder, P.; McGrew, B.; Amodei, D.; McCandlish, S.; Sutskever, I.; Zaremba, W. Evaluating Large Language Models Trained on Code, 2021, [arXiv:cs.LG/2107.03374].
27. Austin, J.; Odena, A.; Nye, M.; Bosma, M.; Michalewski, H.; Dohan, D.; Jiang, E.; Cai, C.; Terry, M.; Le, Q.; Sutton, C. Program Synthesis with Large Language Models, 2021, [arXiv:cs.PL/2108.07732].
28. Liu, J.; Xia, C.S.; Wang, Y.; Zhang, L. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. Proceedings of the 37th International Conference on Neural Information Processing Systems; Curran Associates Inc.: Red Hook, NY, USA, 2024; NIPS '23.
29. Gu, A.; Rozière, B.; Leather, H.; Solar-Lezama, A.; Synnaeve, G.; Wang, S.I. CRUXEval: A Benchmark for Code Reasoning, Understanding and Execution, 2024, [arXiv:cs.SE/2401.03065].
30. Zheng, Q.; Xia, X.; Zou, X.; Dong, Y.; Wang, S.; Xue, Y.; Shen, L.; Wang, Z.; Wang, A.; Li, Y.; Su, T.; Yang, Z.; Tang, J. CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Benchmarking on HumanEval-X. Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining; Association for Computing Machinery: New York, NY, USA, 2023; KDD '23, p. 5673–5684. doi:10.1145/3580305.3599790.
31. Cassano, F.; Gouwar, J.; Nguyen, D.; Nguyen, S.; Phipps-Costin, L.; Pinckney, D.; Yee, M.H.; Zi, Y.; Anderson, C.J.; Feldman, M.Q.; Guha, A.; Greenberg, M.; Jangda, A. MultiPL-E: A Scalable and Extensible Approach to Benchmarking Neural Code Generation, 2022, [arXiv:cs.LG/2208.08227].
32. Athiwaratkun, B.; Gouda, S.K.; Wang, Z.; Li, X.; Tian, Y.; Tan, M.; Ahmad, W.U.; Wang, S.; Sun, Q.; Shang, M.; Gonugondla, S.K.; Ding, H.; Kumar, V.; Fulton, N.; Farahani, A.; Jain, S.; Giaquinto, R.; Qian, H.; Ramanathan, M.K.; Nallapati, R.; Ray, B.; Bhatia, P.; Sengupta, S.; Roth, D.; Xiang, B. Multi-lingual Evaluation of Code Generation Models, 2023, [arXiv:cs.LG/2210.14868].
33. Matton, A.; Sherborne, T.; Aumiller, D.; Tommasone, E.; Alizadeh, M.; He, J.; Ma, R.; Voisin, M.; Gilsenan-McMahon, E.; Gallé, M. On Leakage of Code Generation Evaluation Datasets, 2024, [arXiv:cs.CL/2407.07565].
34. White, C.; Dooley, S.; Roberts, M.; Pal, A.; Feuer, B.; Jain, S.; Shwartz-Ziv, R.; Jain, N.; Saifullah, K.; Naidu, S.; Hegde, C.; LeCun, Y.; Goldstein, T.; Neiswanger, W.; Goldblum, M. LiveBench: A Challenging, Contamination-Free LLM Benchmark, 2024, [arXiv:cs.CL/2406.19314].
35. Liu, C.; Zhang, S.D.; Ibrahimzada, A.R.; Jabbarvand, R. CodeMind: A Framework to Challenge Large Language Models for Code Reasoning, 2024, [arXiv:cs.SE/2402.09664].
36. Malfa, E.L.; Weinhuber, C.; Torre, O.; Lin, F.; Marro, S.; Cohn, A.; Shadbolt, N.; Wooldridge, M. Code Simulation Challenges for Large Language Models, 2024, [arXiv:cs.LG/2401.09074].
37. Dinella, E.; Chandra, S.; Maniatis, P. CRQBench: A Benchmark of Code Reasoning Questions, 2024, [arXiv:cs.SE/2408.08453].
38. Ma, L.; Liu, S.; Bu, L.; Li, S.; Wang, Y.; Liu, Y. SpecEval: Evaluating Code Comprehension in Large Language Models via Program Specifications, 2024, [arXiv:cs.SE/2409.12866].
39. Chen, J.; Pan, Z.; Hu, X.; Li, Z.; Li, G.; Xia, X. Reasoning Runtime Behavior of a Program with LLM: How Far Are We?, 2024, [arXiv:cs.SE/2403.16437].
40. Du, X.; Liu, M.; Wang, K.; Wang, H.; Liu, J.; Chen, Y.; Feng, J.; Sha, C.; Peng, X.; Lou, Y. ClassEval: A Manually-Crafted Benchmark for Evaluating LLMs on Class-level Code Generation, 2023, [arXiv:cs.CL/2308.01861].
41. Jain, N.; Han, K.; Gu, A.; Li, W.D.; Yan, F.; Zhang, T.; Wang, S.; Solar-Lezama, A.; Sen, K.; Stoica, I. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code, 2024, [arXiv:cs.SE/2403.07974].

42. Manh, D.N.; Chau, T.P.; Hai, N.L.; Doan, T.T.; Nguyen, N.V.; Pham, Q.; Bui, N.D.Q. CodeMMLU: A Multi-Task Benchmark for Assessing Code Understanding Capabilities of CodeLLMs, 2024, [arXiv:cs.SE/2410.01999].
43. Lin, D.; Koppel, J.; Chen, A.; Solar-Lezama, A. QuixBugs: a multi-lingual program repair benchmark set based on the quixey challenge. Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity; Association for Computing Machinery: New York, NY, USA, 2017; SPLASH Companion 2017, p. 55–56. doi:10.1145/3135932.3135941.
44. Puri, R.; Kung, D.S.; Janssen, G.; Zhang, W.; Domeniconi, G.; Zolotov, V.; Dolby, J.; Chen, J.; Choudhury, M.; Decker, L.; Thost, V.; Buratti, L.; Pujar, S.; Ramji, S.; Finkler, U.; Malaika, S.; Reiss, F. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks, 2021, [arXiv:cs.SE/2105.12655].
45. Khan, M.A.M.; Bari, M.S.; Do, X.L.; Wang, W.; Parvez, M.R.; Joty, S. xCodeEval: A Large Scale Multilingual Multitask Benchmark for Code Understanding, Generation, Translation and Retrieval, 2023, [arXiv:cs.CL/2303.03004].
46. Zhao, Y.; Luo, Z.; Tian, Y.; Lin, H.; Yan, W.; Li, A.; Ma, J. CodeJudge-Eval: Can Large Language Models be Good Judges in Code Understanding?, 2024, [arXiv:cs.SE/2408.10718].
47. Xu, R.; Cao, J.; Lu, Y.; Lin, H.; Han, X.; He, B.; Cheung, S.C.; Sun, L. CRUXEval-X: A Benchmark for Multilingual Code Reasoning, Understanding and Execution, 2024, [arXiv:cs.AI/2408.13001].