CoderUJB: An Executable and Unified Java Benchmark for Practical Programming Scenarios

Zhengran Zeng
Peking University
Beijing, China
zhengranzeng@stu.pku.edu.cn

Yidong Wang
Peking University
Beijing, China
yidongwang37@gmail.com

Rui Xie*
Peking University
Beijing, China
ruixie@pku.edu.cn

Wei Ye*
Peking University
Beijing, China
wye@pku.edu.cn

Shikun Zhang*
Peking University
Beijing, China
zhangsk@pku.edu.cn

ABSTRACT

In the evolving landscape of large language models (LLMs) tailored for software engineering, the need for benchmarks that accurately reflect real-world development scenarios is paramount. Current benchmarks are either too simplistic or fail to capture the multi-tasking nature of software development. To address this, we introduce CoderUJB, a new benchmark designed to evaluate LLMs across diverse Java programming tasks that are executable and reflective of actual development scenarios, acknowledging Java's prevalence in real-world software production. CoderUJB comprises 2,239 programming questions derived from 17 real open-source Java projects and spans five practical programming tasks. Our empirical study on this benchmark investigates the coding abilities of various open-source and closed-source LLMs, examining the effects of continued pre-training in specific programming languages code and instruction fine-tuning on their performance. The findings indicate that while LLMs exhibit strong potential, challenges remain, particularly in non-functional code generation (e.g., test generation and defect detection). Importantly, our results advise caution in the specific programming languages continued pre-training and instruction fine-tuning, as these techniques could hinder model performance on certain tasks, suggesting the need for more nuanced strategies. CoderUJB thus marks a significant step towards more realistic evaluations of programming capabilities in LLMs, and our study provides valuable insights for the future development of these models in software engineering.

CCS CONCEPTS

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

*Rui Xie, Wei Ye, and Shikun Zhang are the corresponding authors.

KEYWORDS

Code Generation, Large Language Models, Benchmark

ACM Reference Format:
Zhengran Zeng, Yidong Wang, Rui Xie, Wei Ye, and Shikun Zhang. 2024. CoderUJB: An Executable and Unified Java Benchmark for Practical Programming Scenarios. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3652115

1 INTRODUCTION

Researchers have found that advanced AI technologies, exemplified by large language models (LLMs), are proficient in addressing a broad spectrum of challenges, spanning from everyday tasks to complex software engineering issues [1, 10, 40, 52–54, 56, 60, 64]. In addition to these general-purpose LLMs, there are code-centric large language models (code LLMs), such as CodeX [10], CodeLlama [44], and StarCoder [27], which are specifically designed to excel at software engineering tasks. Many of these code LLMs are open-source and can be privately deployed to avoid data security issues. As a result, they have gained significant attention for their strong coding skills. Because of this interest, different benchmarks [10, 13, 16, 23, 30, 59] have been designed to measure the programming capabilities of these LLMs. Specifically, Table 1 presents a selection of notable benchmarks within software engineering alongside our CoderUJB, illustrating their distinct characteristics. The HumanEval [10] benchmark stands out in this field, which is used to evaluate the ability of Python function generation and consists of 164 manually designed Python programming questions, and evaluates the quality of the generated solutions by checking whether the solution can be successfully executed by unit tests. We denote those execution-based evaluations as "**Executable (✓)**". Then, CoderEval [59] noticed that the questions in HumanEval are simple single-function generation tasks that do not match the actual development scenarios (i.e., writing code in a software project). So, they introduced a new benchmark with 460 questions that better aligned with the actual development scenarios. We denote those actual development questions as "**Project-Runnable (✓)**", indicating that the generated code requires a project context dependency for execution. However, those benchmarks focus on a single programming scenario and cannot provide a comprehensive

Table 1: Comparison of existing code benchmark and CoderUJB.

Benchmark	Questions	Executable	Project-Runnable	Multi-Tasks
HumanEval [10]	164	✓	×	×
MultiPL-E [9]	2,952	✓	×	×
MBPP [7]	974	✓	×	×
CoderEval [59]	460	✓	✓	×
Defects4j [21]	835	✓	✓	×
ChatTester [62]	1,000	✓	✓	×
Libro [22]	750	✓	✓	×
CodeXGLUE [30]	759,000	×	×	✓
XCodeEval [23]	159,464	✓	×	✓
CoderUJB	2,239	✓	✓	✓

evaluation of the programming capability of LLMs. Previous multi-task benchmarks, like CodeXGLUE [30], have been critiqued [10] because they rely on similarity-based metrics like BLEU and Code-BLEU [43], which do not involve running the code. The recently proposed multi-programming task dataset XCodeEval [23] focuses on questions from programming competitions, which do not accurately reflect typical real-world development scenarios. Consequently, the absence of a benchmark that comprehensively covers multi-programming tasks, executability, and matches real-world development scenarios prevents us from assessing the effectiveness of current LLMs on a broader range of real-world programming tasks.

To this end, we introduce CoderUJB, a comprehensive benchmark designed for evaluating LLMs that supports multiple tasks, adheres to real-world software development scenarios, and allows for execution within a complete program context(i.e., all source code and execution environments). Specifically, CoderUJB is built on 17 real open-source Java projects, acknowledging Java's prevalence in real-world software production [6]. We extracted 238 functional code generation questions and 140 code-based test generation [62] questions from these projects by analyzing the abstract syntax trees and test coverage relationships of the project source code. Then, we extracted and collected 451 issue-based test generation [22] questions, 470 automatic program repair [56] questions, and 940 defect detection [66] questions from the projects by combining the detailed defect information and related issue reports from the projects. Altogether, CoderUJB comprises 2,239 programming questions covering five trending and practical programming tasks, which is the largest benchmark that is "Project-Runnable" and each question comes with a complete program context to facilitate the researcher's detailed analysis of the questions and the generated solutions. Ultimately, the solutions generated by the LLMs will be placed in real projects for execution, and the programming ability of the LLMs will be evaluated using the execution success rate as the primary metric.

Next, to illustrate the value of CoderUJB to the field, we conducted a comprehensive empirical study on CoderUJB to explore the programming abilities of a representative set of open-source code LLMs and general-purpose closed-source LLMs. The aim was to answer a couple of crucial questions: How good are these LLMs at coding? And how do continued pre-training [44] and instruction fine-tuning [31] affect their programming performance? After running a slew of experiments on CoderUJB, we find that current LLMs still perform poorly in solving non-functional code generation tasks, especially defect detection tasks. Secondly, superior open-source LLMs can already approach or even surpass GPT-3.5-Turbo on the functional code generation task and the two test generation tasks. Thirdly, continued pre-training in a specific programming language can reduce an LLM's performance in other languages, and this negative impact diminishes as the task becomes less related to the original pre-training task. Such varied outcomes across different tasks emphasize CoderUJB's value as a unified evaluation benchmark that incorporates multiple programming tasks. Lastly, our findings indicate that instruction fine-tuning diminishes the performance of code LLMs in functional code generation and the two test generation tasks—a contrast to the results from the less practical benchmark, HumanEval. This highlights the value of CoderUJB as a practical programming evaluation benchmark.

We summarize the main contributions of this study as follows:

- **Benchmark.** We have introduced CoderUJB, a universal Java benchmark for assessing LLM performance across multiple real-world programming tasks. The benchmark includes executable test cases and the complete program context (i.e., all source code and execution environments), comprising 2,239 programming questions.
- **Study.** We ran a comprehensive empirical analysis of both open-source and proprietary LLMs using CoderUJB, studying (1) their performance across diverse coding tasks, (2) the effect of continued pre-training in a specific language code, and (3) the impact of instruction fine-tuning on these models.
- **Implications.** This work revealed multiple significant findings: (1) Program context is useful in code generation tasks. (2) Caution is advised when continuing pre-training in a specific programming language, as its effects tend to be unpredictable, especially if the downstream task is substantially different from the pre-training task. (3) Instruction fine-tuning should be approached carefully, as it can diminish the performance of LLMs on tasks that align closely with the pre-trained task. (4) Comprehensive evaluations are essential, given that different programming tasks may yield disparate results when the same training strategy is applied.

The CoderUJB are publicly available in **https://github.com/WisdomShell/ujb**.

2 BACKGROUND AND RELATED WORK

2.1 Large Language Models for Software Engineering

Recently, many code LLMs [10, 27, 37, 44] have been proposed to solve software engineering tasks. These models typically leverage the Transformer Decoder [50] architecture and undergo extensive training on large-scale code databases [10, 27, 37], expecting excellence in executing code-intensive tasks.

A prominent example within this realm is CodeX [10]. For the first time, they increased the model parameter size to 12 billion (B) and used 159 gigabytes (GB) of Python code data for pre-training. They also proposed the widely adopted benchmark HumanEval and the *pass@k* metric for evaluating the quality of generated code, in which CodeX achieved a *pass@k*=1 of 28.81 on HumanEval. Subsequently, StarCoderBase [27] further scaled both the parameter size to 15B and the training dataset to 799.37 GB code corpus, which contained various programming languages. We designate those LLMs that have not undergone specialized training for particular tasks (e.g., Python programming challenges or question-and-answer activities) as "**Base LLM**".

As mentioned, to enhance Python-specific performance on tasks like HumanEval, researchers continued pre-training the base model StarCoderBase on an additional 35B tokens of Python code, yielding a Python-enhanced version of the model StarCoder-Python. Ultimately, StarCoderBase and StarCoder-Python achieved a *pass@k*=1 of 30.4 and 33.6 on HumanEval, respectively. We designate those LLMs that have continued pre-training in specific programming language corpus as "**Specific Programming Language (PL) Base**".

Moreover, the team behind WizardCoder [31] found that fine-tuning base LLMs with high-quality instruction samples (i.e., more diverse and more challenging coding problems) further boosted the LLMs programming performance. They used an evolutionary instruction-based data collection strategy to collect 80k instruction fine-tuning samples from ChatGPT. Utilizing these novel datasets, they fine-tuned a "**Instruction Tuned**" LLM WizardCoder, significantly enhancing the HumanEval *pass@k*=1 rate from 33.57 to an impressive 58.12.

Despite the progress, the prevailing methods of evaluating these code LLMs tend to focus on simple Python-based programming puzzles like HumanEval and lack a comprehensive assessment of other programming tasks in software engineering. This limited evaluation does not fully capture the advancements made in code LLMs, nor does it comprehensively evaluate the impact of specialized training processes such as continued pre-training in a specific language code (mostly in Python) and instructional fine-tuning on various practical software engineering scenarios. In response to these limitations, our study introduces a multi-programming task, executable and real-world programming scenario-compliant evaluation benchmark, and provides an in-depth study to address the questions mentioned above.

2.2 Coder Benchmark

Recent scholarly efforts have proposed many benchmarks [7, 9, 10, 23, 59] to evaluate the programmatic skill of LLMs as presents in Table 1. The HumanEval [10] benchmark stands out in this field, consisting of 164 manually designed Python programming questions. Each question provides function signatures, comments, function bodies, and multiple unit tests. LLMs are tasked with crafting function bodies informed by the given signatures and comments. Subsequently, the generated functions are executed and evaluated regarding their success in passing the unit tests tied to each question. We refer to such non-test-case code generation questions as functional code generation tasks. Later, CoderEval noticed that the questions in HumanEval are simple "self-contained" [59] function generation tasks that each function only has language built-in dependency and do not match the actual development scenarios that rely on multiple public libraries and project files. Therefore, they introduced 460 Java and Python code generation questions derived from real projects on GitHub that are more aligned with actual development settings.

In addition to mainstream functional code generation tasks, there are other types of datasets designed to evaluate LLMs. For example, Defects4j [21], a seminal benchmark in automated program repair, catalogs 835 defects across 17 authentic Java projects from Github. It offers both the defective and fixed versions of the code alongside related test cases. In this evaluative phase, the fix code generated by LLMs is executed, and the accompanying test cases verify the correctness of the fix. Based on Defects4j, Libro [22] has developed a benchmark for generating issue-specific tests. This requires LLMs to generate test cases that trigger the corresponding defects based on a given issue report. Additionally, the ChatTester [62] benchmark, designated for code-based test generation tasks, collects 1,000 test generation questions from open-source Java projects on GitHub. Each test question contains the code needed for testing, a natural language description of the task, and a test case. LLMs need to generate test cases based on the code under test and the natural language description.

Besides the single-task benchmark mentioned above, previous researchers have proposed two multi-programming task benchmarks, CodeXGLUE [30] and XCodeEval [23], that incorporate a broad range of questions and tasks, establishing a more substantial framework for evaluating LLMs. However, CodeXGLUE still uses textual similarity metrics, such as BLEU and CodeBLEU [43], thereby falling short of actual code execution. XCodeEval focuses on programming competition questions, which, like HumanEval, are not "Project-Runnable" and do not match the actual development scenarios. Therefore, CoderUJB fills the current gap in the field as a benchmark that simultaneously includes multiple programming tasks that are executable and match real-world development scenarios, thus broadening the scope of evaluation for LLMs across various practical programming tasks.

3 CODERUJB BENCHMARK

This section outlines the programming tasks included in the Coder -UJB and provides an overview of the dataset construction process and evaluation metrics.

3.1 Tasks in CoderUJB

CoderUJB has chosen four representative program generation tasks and one crucial program classification task to create a comprehensive and uniform dataset for evaluating programming skills.

Functional Code Generation (FCG): Code generation is a crucial topic for LLMs in software development [10, 30], often consuming much of the developer's time. In this task, we need to generate the corresponding function code based on the given function annotations. We refer to previous work [10, 59] to define the input of the task as function-related context, function annotations, and function signatures, while the output of the task is a function that meets the task requirements, and test cases are used to evaluate the quality of the generated function. Note that this task is limited to generating operational code that executes the service's logic (i.e., not test case function or configuration file).

Code-based Test Generation (CTG): Test generation is another part of the code generation task, but it requires programming skills different from functional code generation. For example, writing test cases usually requires more cross-functional understanding, whereas generating functional functions tends to focus only on the content of the current function following decoupled design principles [51]. Therefore, we refer to previous work [62] on test generation as a separate task. This task involves reading and understanding the program logic of the code under test and then generating test cases that verify the code's core functionality based on the test cases' function annotations. Consequently, we define the input of the task as task-related context, test case annotations and test case signatures, and the output of the task as test cases that meet the task requirements, and evaluate the quality of the generated test cases based on metrics such as whether they meet the task requirements and test coverage, which will be detailed in Section 3.5.

Issue-based Test Generation (ITG): Issue report-based test generation is a process where LLMs analyze and comprehend bugs reported in issue reports. Following the comprehension, they devise test cases to reproduce those issues. We follow the task definition developed in previous works [22] for this kind of task. The task input consists of two elements: an issue report detailing a particular bug and the corresponding function signature linked to that bug, while the output of the task is defined as the test case that meets the task requirements.

Defect Detection (DD): Defect detection is a critical activity in software engineering, significantly influencing the overall software development process [28, 66]. The primary purpose of this task is to detect possible bugs in the software's code, including, but not limited to, syntax errors and potential runtime errors. Following the previous work's task definition [66], our task input for defect detection would include the specific function under scrutiny, and the output is a statement that indicates whether or not a bug exists.

Automated Program Repair (APR): After successfully identifying issues within a program, the next step is to make corrections. In the task of automated program repair, LLMs are given samples of functions containing defects, and they are expected to fix these errors, returning functions that are free of defects. This task is crucial in software engineering, so we align our definition with previous task definitions [56]. In this case, the task input includes task-related contexts and defective functions; the output should be fixed functions. Eventually, test cases are used to evaluate the correctness of the fixed functions.

3.2 Datasets in CoderUJB

This section introduces the process of collecting and processing CoderUJB questions. Figure 1 illustrates the overview of the data processing procedure within CoderUJB. As mentioned in Section 2.2, Defects4j [21] provided 17 qualitative and practical open source Java projects as well as their defects data and issue reports, due to its completeness and quality, it has become the infrastructure in the field of automated program repair [19, 32, 56]. Recognizing the value of Defects4j, we build a comprehensive benchmark on it. Consequently, each of the five programming tasks of CoderUJB gleaned its source data from 17 projects under Defects4j. Additionally, while it could incorporate other sources to build CodeUJB. we chose to collecting our coding problems across different tasks from the same reliable source (i.e., Defects4j), providing similar quality and difficulty levels across tasks and minimizing data quality impacts on our findings. Such selection offers substantial benefits over using varied datasets from disparate sources.

3.3 Dataset Construction

Functional Code Generation (FCG): In the processing flow for the functional code generation benchmark, we first select the latest defect-free versions of the 17 open-source projects within Defects4j [21] as the original project repository. We opt for the latest versions because they undergo continuous evolution and development, thus likely possessing the fewest potential defects and demonstrating superior code quality. Subsequently, we extract all functions and test cases found in the project through an abstract syntax tree (AST) investigation. Then, as depicted in the "`Analysis and Filtering Process`" in Figure 1, we first analyze the coverage relationship between test cases and functions via a test coverage analysis tool [3] to determine the test cases calling each function. Following this, we gather question-related data about each function, which includes the function body, comments, source code of the associated class, and related test cases. Later, we eliminated low-quality data by referring to CoderEval's guidelines [59]. More specifically, we ensure that: 1) The function is not a test, interface, or deprecated. 2) The function has function-level comment in English. 3) The test coverage ratio larger than 50%. Then, the question units are ranked in descending order based on the associated test coverage ratio, and a manual quality assessment is performed to further omit low-quality functions, such as oversimplified getter and setter functions. Ultimately, 238 functional code generation questions were filtered out, and each function had about 161.66 test cases on average. The question prompt is then formulated, and the exact structure will be discussed in detail in Section 3.4.

Code-based Test Generation (CTG): For the code-based test generation benchmark, we also select the latest defect-free project version in Defects4j as the initial project repository. We then follow the same process for the functional code generation dataset to extract functions and test cases, while also examining their coverage relationship. We subsequently filter out low-quality data using standards similar to those applied by CoderEval [59] and ChatTester [62]. More specifically, we ensure that the test case 1) has name includes a test-associated keyword such as "Test", 2) is not deprecated, 3) has function-level comments in English, 4) is in a class that corresponds to a related functional class, such as "Example.java" and "ExampleTest.java". Finally, we manually evaluate their quality to eliminate any low-quality instances, such as those with low test coverage, which are usually the bug reproduced test cases, and the ability to generate such test-cases we would like to examine in the following issue-based test generation task. We finally filtered out 140 code-based test cases to generate the test questions.

Automated Program Repair (APR): As previously noted in Section 2.2, the Defects4j [21] dataset is a prevalent source for automated program repair (APR). In designing our benchmark for APR, we directly adopt the defect dataset provided by Defects4j and refer to the previous work [56] to focus only on the single-function defects in Defects4j. This specific focus makes it easier for us to create a uniform question prompt, which ultimately enhances

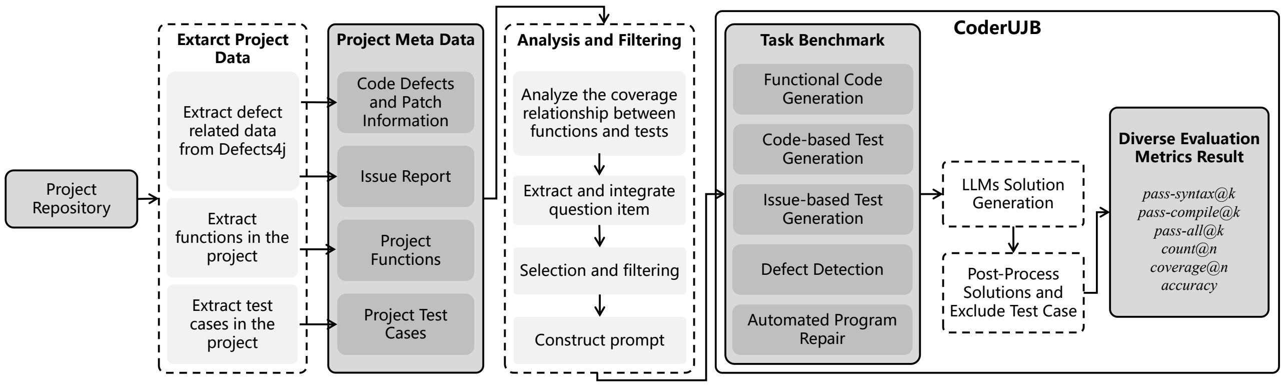

Project Repository
Extarct Project Data
Extract defect related data from Defects4j
Extract functions in the project
Extract test cases in the project
Project Meta Data
Code Defects and Patch Information
Issue Report
Project Functions
Project Test Cases
Analysis and Filtering
Analyze the coverage relationship between functions and tests
Extract and integrate question item
Selection and filtering
Construct prompt
CoderUJB
Task Benchmark
Functional Code Generation
Code-based Test Generation
Issue-based Test Generation
Defect Detection
Automated Program Repair
LLMs Solution Generation
Post-Process Solutions and Exclude Test Case
Diverse Evaluation Metrics Result
pass-syntax@k
pass-compile@k
pass-all@k
count@n
coverage@n
accuracy

Figure 1: Overview of CoderUJB construction process.

the quality of our benchmark questions. In the end, we extracted 470 single-function defects from the dataset to serve as our APR benchmark.

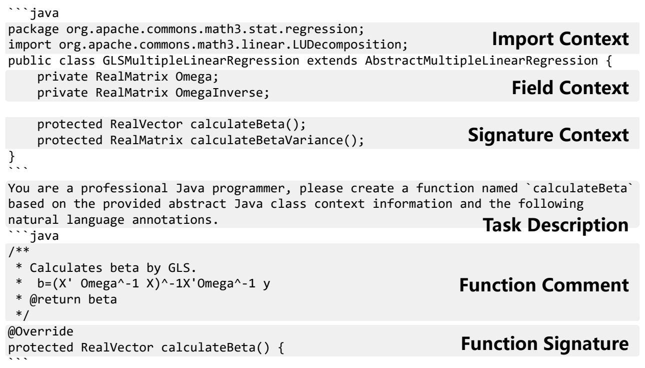
Defect Detection (DD): For the defect detection benchmark, we have also developed a benchmark based on the Defects4j. Specifically, we focused on single-function defects as well, selecting 470 such bugs in Defects4j and using their defective functions along with their corresponding fixed versions as positive and negative samples. In the end, we have compiled a balanced dataset containing 940 detection samples for defect detection.

Issue-based Test Generation (ITG): As mentioned in Section 2.2, LIBRO [22] has proved a comprehensive benchmark for issue-based test generation based on Defects4j. Therefore, from the 750 test generation questions provided by LIBRO, we kept those related to issues that match the bugs in our automated program repair benchmark. Consequently, we obtained a benchmark consisting of 451 issue-based test generation questions.

3.4 Base Prompt Design

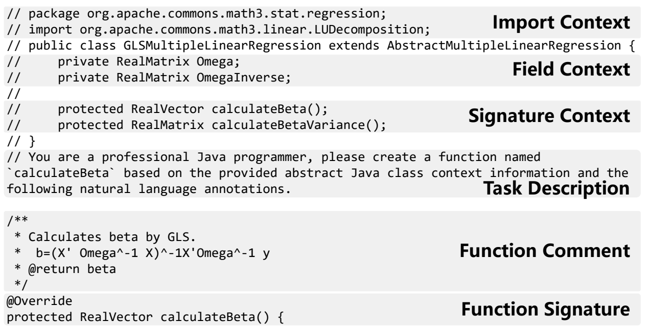
Functional Code Generation (FCG): Figure 2 shows an example prompt for a functional code generation task consisting of six parts. Among them, "Import Context", "Filed Context", and "Signature Context" serve as task-related contexts, extracted from the corresponding Java files of the task functions through AST analysis. We will show the effectiveness of those contexts in Section 4.3.1. The other three components, namely "Task Description", "Function Comment", and "Function Signature", define the specific requirements of the task, instructing the LLMs on the desired code content to be generated.

It is important to note that the prevalent usage of current LLMs falls into two ways: chat invocation (e.g., ChatGPT [1] with chat alignment) and complement invocation (e.g., StarCoderBase [27] without chat alignment). However, most users prefer the chat invocation method. Additionally, most open-source LLMs are typically offered only in the base version (i.e., without fine-tuning) [27, 37, 49], limiting us to using the model in a complement invocation way. For a fair comparison of these two types of LLMs, we have proposed two invocation prompts for each task: one in the style of a chat (i.e., Figure 2a) and the other in a complementary format (i.e., Figure 2b). While the content of the two prompts remains the same, they differ in how the information is formatted. Hence, we


Import Context
Field Context
Signature Context
Task Description
Function Comment
Function Signature

~~~
```java
package org.apache.commons.math3.stat.regression;
import org.apache.commons.math3.linear.LUDecomposition;
public class GLSMultipleLinearRegression extends AbstractMultipleLinearRegression {
    private RealMatrix Omega;
    private RealMatrix OmegaInverse;

    protected RealVector calculateBeta();
    protected RealMatrix calculateBetaVariance();
}
```
You are a professional Java programmer, please create a function named `calculateBeta`
based on the provided abstract Java class context information and the following
natural language annotations.
```java
/**
 * Calculates beta by GLS.
 *  b=(X' Omega^-1 X)^-1X'Omega^-1 y
 * @return beta
 */
@Override
protected RealVector calculateBeta() {
```
~~~

(a) Prompt of chat invocation.


Import Context
Field Context
Signature Context
Task Description
Function Comment
Function Signature

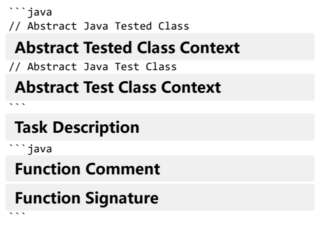
```
// package org.apache.commons.math3.stat.regression;
// import org.apache.commons.math3.linear.LUDecomposition;
// public class GLSMultipleLinearRegression extends AbstractMultipleLinearRegression {
//     private RealMatrix Omega;
//     private RealMatrix OmegaInverse;
//
//     protected RealVector calculateBeta();
//     protected RealMatrix calculateBetaVariance();
// }
// You are a professional Java programmer, please create a function named
`calculateBeta` based on the provided abstract Java class context information and the
following natural language annotations.

/**
 * Calculates beta by GLS.
 *  b=(X' Omega^-1 X)^-1X'Omega^-1 y
 * @return beta
 */
@Override
protected RealVector calculateBeta() {
```

(b) Prompt of complement invocation.

Figure 2: Prompt of Functional Code Generation.

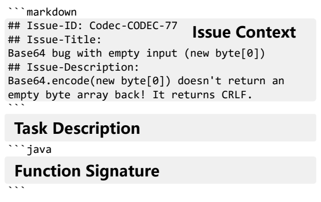
consider these two prompts equivalent, enabling a fair comparison between the two invocation ways.

Code-based Test Generation (CTG): Figure 3a illustrates a chat invocation prompt for code-based test generation. In the task-related context, we provide both the under-test class context (i.e., the "Abstract Tested Class Context" in Figure 3a) and the test class context (i.e., the "Abstract Test Class Context" in Figure 3a). The term "Abstract Class Context" here indicates the aggregate of "Import Context", "Field Context", and "Signature Context" in Figure 2. Additionally, the task prompt also has a "Task Description", "Function Comment", and "Function Signature" to outline the unique requirements of the task. Meanwhile, the



~~~~
```java
// Abstract Java Tested Class
Abstract Tested Class Context
// Abstract Java Test Class
Abstract Test Class Context
```
Task Description
```java
Function Comment
Function Signature
```
~~~~

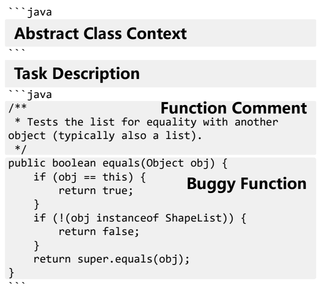
(a) Chat prompt of Code-based Test Generation.



~~~~
```markdown
## Issue-ID: Codec-CODEC-77
## Issue-Title:
Base64 bug with empty input (new byte[0])
## Issue-Description:
Base64.encode(new byte[0]) doesn't return an
empty byte array back! It returns CRLF.
```
Task Description
```java
Function Signature
```
~~~~

Issue Context

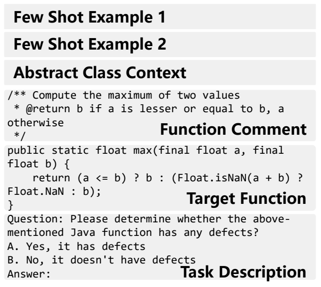
(b) Chat prompt of Issue-based Test Generation.



~~~~
```java
Abstract Class Context
```
Task Description
```java
/**
 * Tests the list for equality with another
object (typically also a list).
 */
public boolean equals(Object obj) {
    if (obj == this) {
        return true;
    }
    if (!(obj instanceof ShapeList)) {
        return false;
    }
    return super.equals(obj);
}
```
~~~~

Function Comment
Buggy Function

(c) Chat prompt of Automated Program Repair.



~~~~
Few Shot Example 1
Few Shot Example 2
Abstract Class Context
/** Compute the maximum of two values
 * @return b if a is lesser or equal to b, a
otherwise
 */
public static float max(final float a, final
float b) {
    return (a <= b) ? b : (Float.isNaN(a + b) ?
Float.NaN : b);
}
Question: Please determine whether the above-
mentioned Java function has any defects?
A. Yes, it has defects
B. No, it doesn't have defects
Answer:
~~~~

(d) Complement prompt of Defect Detection.

Figure 3: Prompt of CoderUJB.

complement invocation prompt for this task is similar to Figure 2a, with the prompt components reformatted and organized.

Issue-based Test Generation (ITG): Figure 3b presents an example chat invocation prompt for issue-based test generation. This prompt format is guided by the LIBRO [22]. More specifically, this task's prompt is composed of three key elements. The first element is the "`Issue Context`"; it includes the issue report featuring the issue ID, title, and a detailed description specific to that issue. The other two elements, "`Task Description`" and "`Function Signature`" lay out the exact requirements of the task. The completion prompt for this task is also with the prompt components reformatted and organized. It is noteworthy that we do not provide "`Abstract Class Context`" and "`Function Comment`" in the prompt because we want to evaluate the capacity of LLMs in deriving those task information directly from issue reports [22].

Automated Program Repair (APR): Figure 3c shows an example chat invocation prompt for an automated program repair task. The design of this prompt references from prior APR work [56]. More specifically, this prompt has four key components and is similar to the design for functional code generation in Figure 2a. The main difference is that the prompt for the APR task includes the "`Buggy Function`", which is the defective function that needs to be fixed by the LLMs.

Defect Detection (DD): Figure 3d presents a complement invocation prompt for defect detection. Considering that this task is categorical, our prompt structure borrows from the design paradigms of classification benchmarks in the natural language processing (NLP) domain, such as MMLU [17] and C-Eval [18], which utilize few-shot examples to guide base LLMs (i.e., not fine-tuned with instructions) in generate valid output (i.e., the option of a multiple-choice question). We have implemented a two-shot format [34], following with relevant code context and a binary multiple-choice question.

3.5 Metrics in CoderUJB

3.5.1 *pass@k.*

We apply the *pass@k* metric to evaluate the generated solutions of the code generation task as it has been widely used in previous researches [10, 14, 59]. Specifically, given an unordered set of n candidate solutions, *pass@k* indicates the probability of selecting at least one correct solution in k solutions sample from all n candidate solutions. We use the following formula to compute *pass@k* defined by previous work [10]:

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

Given that the number of generated solutions is n, the number of solutions used to estimate *pass@k* is k, and c is the number of correct solutions out of n samples. Moreover, the *pass@k* result for the entire dataset is the expected value (mean) of the *pass@k* for individual problems. It is important to note that we have set n to 20 in our study (doubling the size used in prior research [59]) to improve the stability of our findings.

In functional code generation and automated program repair scenarios, a solution is correct if it passes all relevant tests. In the test generation scenario, we refer to the definition of previous work [62] and define a solution to be correct if and only if the test case can be run successfully and covers the tested code. In addition, in the issue-based test generation scenario, the test case must reproduce the issue bug by reporting the error in the defective project version but running successfully in the fixed version [22]. Beyond executing correct metrics, we recorded the outcomes at various executing stages to better examine the solution. Specifically, we employ a *pass@k* metric for syntax checking, defined as *pass-syntax@k*, where correctness equates to successful syntax checking, and another metric used for compilation checking, referred to as *pass-compile@k*, where we consider the solution to be correct if it compiles successfully. Lastly, we use a metric, *pass-all@k*, for situations where all test cases and checklists are passed.

3.5.2 *count@n.*

To demonstrate the ability of LLMs to solve code-generation problems more intuitively, we introduce a metric *count-@n*, which is similar to the standard metric *plausiblepatches* in the field of APR [14, 19, 32, 56]. Specifically, this metric measures the number of coding problems an LLM can successfully solve by generating n solutions for each problem. Therefore, the *count@n* value assigned to each problem is defined as:

$$count@n := \bigvee_{i=1}^{n} correct(solution_i) \quad (2)$$

If any of the n solutions generated by the LLMs for a given problem is correct, the *count@n* value for that problem is assigned as 1, and for all other situations, it would be 0. The *count@n* score across an entire benchmark is the sum of *count@n* values for each problem within the benchmark.

3.5.3 *coverage@n.*

Test coverage is a crucial metric for measuring the effectiveness of test cases [51]. Therefore, we also counted the combined test coverage for the code under test for the test generation task. Specifically, for each set of n solutions generated for a particular programming problem, we first identify and record the lines of code that each solution can cover. Subsequently, we accumulate the code lines covered from all *n* solutions into a comprehensive set. The final step is to compute the percentage of this comprehensive set's lines of code against the total lines of code under test. The *coverage@n* of the entire benchmark is then derived as the mean of the *coverage@n* for each problem within the benchmark. Consequently, we define the *coverage@n* metric as follow:

$$coverage@n := \frac{count(\bigcup_{i=1}^{N} cover\_line(solution_i))}{all\_tested\_line\_count} \quad (3)$$

The *cover_line* function can get the lines of code that a specific solution $solution_i$ can cover, and *all_tested_line* indicates the number of lines of all tested code in the programming problem.

3.5.4 accuracy. For the classification task of defect detection, we simply adopt the widely used *accuracy* metrics [17, 18, 30].

4 THE EXTENSIVE STUDY

In this section, we further evaluate existing leading LLMs with CoderUJB to delve into issues that are pertinent to researchers and to showcase CoderUJB's contribution to advancing the field.

4.1 Research Questions

This study investigates the following research questions:

- **RQ1:** *Does the basic program context prompt improve the performance of LLMs on CoderUJB?* To this research question, we aim to explore how various prompts influence the performance of LLMs to confirm whether the basic program context given by CoderUJB is beneficial.
- **RQ2:** *How do open-source and closed-source LLMs perform under CoderUJB?* We are looking to better understand the current progress of two types of LLMs and provide a thorough evaluation of their effectiveness in handling real programming tasks.
- **RQ3:** *How does continued pre-training of specific programming language (PL) data affect the performance of code LLMs under CoderUJB?* This part of our study will explore how continued pre-training of specific programming language data might impact the performance of code LLMs when handling real programming tasks in other coding languages.
- **RQ4:** *How does instruction fine-tuning influence the performance of code LLMs in CoderUJB?* This research question will investigate the potential benefits of instruction fine-tuning strategies on the performance of code LLMs under CoderUJB to provide feasible guidelines for the practical application of code LLMs.

4.2 Code LLMs Subjects

For the study subject, we focus on the widely used code LLMs and three closed-source commercial LLMs. Table 2 provides an overview of these selected models. Specifically, take CodeLlama-7B as an example, "Trained From" and "Training Dataset" represent CodeLlama-7B is trained from Llama2-7B with 500B tokens code corpus and 20B tokens long context corpus. We have classified these LLMs into four primary categories:

Base LLMs: This type of LLM consists of the widely adopted CodeLlama-7B, 13B, 34B [44], and StarCoderBase-15B [27]. We chose these models because they each come with their own Specific-PL-Base and Instruction-Tuned versions, aiding our future comparison studies and experiments. Meanwhile, previous studies [31, 45, 58, 61] have thoroughly researched these models, making them noteworthy representatives of code LLMs. Additionally, we add another CodeShell [57] as a baseline for LLM with lower training resources, as it is trained from scratch using only 500B code tokens. We apply complement prompt for interacting with those LLMs.

Table 2: Statistical information of the studied LLMs.

Type	Model Name	Size (B)	Trained From	Training Dataset
Base	CodeLlama	7;13;34	Llama2	520B code tokens
	StarCoder-Base	15	From Scratch	1T code tokens
	CodeShell	7	From Scratch	500B code tokens
Specific PL Base	CodeLlama-Python	7;13;34	CodeLlama	120B Python tokens
	StarCoder-Python	15	StarCoder-Base	35B Python tokens
	StarCoder-Java	15	StarCoder-Base	35B Java tokens
Instruction Tuned	CodeLlama-Instruct	7;13;34	CodeLlama	5B instruction tokens
	WizardCoder-Python	7;13;34	CodeLlama-Python	80k instructions
	WizardCoder	15	StarCoder-Python	80k instructions
	CodeShell-Chat	7	CodeShell	40k instructions
Closed Source	Claude-1	/	/	/
	GPT-3.5-Turbo	/	/	/
	GPT-4	/	/	/

Specific-PL-Base LLMs: In addition to the Base LLMs, many existing code LLMs [27, 37, 44] have an additional version that undergoes further pre-training on Python data to improve the performance of the base LLM under Python programming tasks. However, this can raise concerns for researchers about how these models would perform with tasks in other programming languages [9, 65]. Therefore, we collected four specific Python base LLMs, namely CodeLlama-Python-7B, 13B, 34B [44], and StarCoder-15B [27]. Moreover, to investigate how continued pre-training on specific programming languages data would affect LLMs on programming tasks in other languages, we follow the setting of StarCoder-Python to further continue pre-training StarCoder-Base on another 35B Java tokens (random sampling from The Stack [24]) and get a specific Java model StarCoder-Java. We apply complement prompt for interacting with those LLMs.

Instruction-Tuned LLMs: Along with the Base LLMs, instruction fine-tuned LLMs are another crucial category of interest [31, 39]. Unlike traditional fine-tuning, which focuses on single-task training, instruction fine-tuning employs diverse task data to train the model. Previous studies [31, 39, 45, 61] have shown that instruction fine-tuning enhances performance across various NLP and programming tasks, showing more promise than traditional fine-tuning strategy. With this in mind, we selected eight open-source, instruction fine-tuned LLMs named CodeLlama-Instruct-7B, 13B, 34B [44], WizardCoder-Python-7B, 13B, 34B, WizardCoder-15B [31], and CodeShell-Chat [57] for our exploration of how instruction fine-tuning influences LLMs in distinct programming tasks. We apply chat prompt for interacting with those LLMs.

Closed-Source LLMs: We also select three closed-source commercial LLMs (i.e., Claude-1 [2], GPT-3.5-Turbo-0301 [1], and GPT-4-0314 [5]) to evaluate the gap between open-source and closed-source LLMs. We apply chat prompt for interacting with those LLMs.

4.3 Results and Analysis

Table 3: Evaluation results of prompt design for CoderUJB.

Task	Metric	StarCoderBase-15B			CodeLlama-13B		
		program context	*one shot*	*four shot*	*program context*	*one shot*	*four shot*
FCG	*pass-all@k*=1	**15.32**	12.18	11.66	**21.91**	12.84	14.50
	pass-all@k=10	**26.82**	19.33	19.66	**34.85**	20.92	22.37
	count-all@n=20	**75**	51	53	**90**	55	57
CTG	*pass-all@k*=1	**12.14**	3.93	5.57	**12.61**	7.04	7.57
	pass-all@k=10	**31.80**	12.24	11.26	**38.17**	17.50	17.69
	count-all@n=20	**52**	24	20	**67**	30	29
	coverage@n=20	**11.91**	6.20	5.39	**15.66**	8.65	8.52
APR	*pass-all@k*=1	**6.56**	4.14	4.64	**4.50**	4.07	4.10
	pass-all@k=10	**12.54**	7.69	8.11	**8.39**	8.20	7.47
	count-all@n=20	**66**	41	44	44	**45**	39
DD	*accuracy*	50.32	**51.19**	48.70	48.60	49.68	**51.51**
	error-count	0	**0**	0	32	**2**	4

4.3.1 RQ1: Does the Basic Program Context Prompt Improve the Performance of LLMs on CoderUJB. To explore whether including program context is beneficial for code LLMs in solving programming tasks, we conducted experiments comparing basic prompts augmented with program context against standard few-shot prompts [34]. These experiments were applied across four different tasks within the CoderUJB. Specifically, the program context prompts for the functional code generation task, the code-based test generation task, and the automatic program repair task are the same as those presented in Section 3.4. For defect detection tasks, the program context prompts were designed based on the prompt of functional code generation. The few-shot prompt examples were chosen from filter-out samples when creating CoderUJB or from other benchmarks [29] of the same task. It is important to note that we did not include an issue-based test generation task as prior research [22] suggests that specific program contexts are not applicable in such scenarios. Due to the page limit, we conducted experiments on two representative open-source code LLMs, CodeLlama-13B and StarCoder-Base-15B, while placing experiments evaluating more LLMs in subsequent sections.

Table 3 shows the results of the prompt comparison experiments. We can find that program context is helpful for most programming tasks. Specifically, in functional code generation and code-based test generation, the program context prompt outperformed the few-shot prompt. For example, StarCoderBase-15B scored 15.32 and 75 in *pass-all@k*=1 and *count-all@n*=20, outperforming the results from other few-shot prompts. Moreover, in code-based test generation, tests generated with program context demonstrated significantly better coverage than those from few-shot prompts. In the automated program repair task, we observed notable improvements in StarCoderBase-15B when using the program context prompt. Therefore, such comparison results clearly show the good quality of the basic prompts from CoderUJB and highlight the value of supplying full program contexts (i.e., all source code and execution environment) in CoderUJB.

However, in the defect detection task, we found that the *accuracy* of prompt with program context is slightly lower than the few-shot prompt. For example, the *accuracy* of CodeLlama-13B with the program context prompt was 48.60%, lower than the 49.68% and 51.51% achieved with the few-shot prompt. We then further show the *error-count* metric of the number of answers that could not be parsed correctly (i.e., not a valid answer), which reveals that the code LLMs may fail to generate valid answers when using

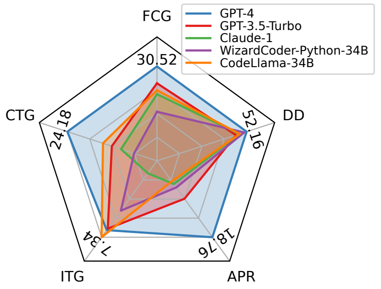


Figure 4: Evaluation results (*accuracy* for DD and *pass-all@k*=1 for others) of open-source LLMs and closed-source LLMs under CoderUJB.

program context prompt, ultimately leading to lower *accuracy*. To capture the impact of invalid answer, we will report both *accuracy* and *error-count* for subsequent experiments. Nevertheless, we are still choosing the program context prompt as the basic prompt for the defect detection task because it achieves similar accuracy to the few-shot prompt, as it also contains two few-shot examples. Additionally, we believe it may perform even better in subsequent experiments since it contains additional program context.

Note that CoderUJB provides a base prompt design that is as reliable as possible. We recognize that these base prompt designs are not best practices. However, this is exactly why we introduced CoderUJB as a comprehensive context (i.e., all source code and execution environment) benchmark. We encourage other researchers to use this benchmark to explore and create even better prompt designs, as there is already a lot of interesting work [36, 40, 63] in this field, and CoderUJB can offer a comprehensive and fair framework for such research.

Conclusion 1: The program context is useful for functional code generation, code-based test generation, and automated program repair.

4.3.2 RQ2: How Do Open-Source and Closed-Source LLMs Perform Under CoderUJB. Table 4 shows the evaluation results of the selected LLMs in five programming scenarios on key performance metrics (i.e., *pass-all@k*, *count-all@n*, *accuracy*). To better demonstrate the performance differences between open-source and closed-source LLMs, we have exhibited the comparison using *pass-all@k* and *accuracy* radar plots in Figure 4, featuring three leading open-source code LLMs and three closed-source LLMs under CoderUJB.

After combining detailed experimental results from Table 4 with Figure 4, it can be found that current LLMs fail to achieve the same impressive results as HumanEval [44] and CoderEval [59] under CoderUJB. Specifically, the most powerful open-source coder LLMs CodeLlama-34B and closed-source LLM GPT-4 can only achieve *pass-all@k*=1 metrics of 22.82 and 30.52 under the functional code generation task, much lower than their results of 45.11 [44] and 67.00 [38] on HumanEval. Their performance under other programming tasks is even worse than that under the functional function generation task, e.g., pass-all@k=1 of GPT-4 under the other 3 code generation tasks are only 24.18, 15.76, and 18.76. Moreover, the CodeShell-7B model, which utilizes limited training resources, falls short in delivering satisfactory results compared to its performance

Table 4: Evaluation results for CoderUJB, HumanEval and CoderEval. *pass-all@k* **denoted as** *p-a=k*, *count-all@n* **denoted as** *c-a=n*, *accuracy* **denoted as** *acc*, *error-count* **denoted as** *err*. **The values in Red indicate underperform, Green values indicate outperform corresponding "Trained From" LLMs as presents in Table 2.**

Type	Model-ID	CoderUJB														HumanEval[1]		CoderEval[2]	
		FCG			CTG			ITG			APR			DD		Java	Py	Java	Py
		p-a@k		*c-a@n*	*p-a@k*		*c-a@n*	*p-a@k*		*c-a@n*	*p-a@k*		*c-a@n*	*acc*	*err*	*p-a@k*		*p-a@k*	
		k=1	*k*=10	*n*=20	*k*=1	*k*=10	*n*=20	*k*=1	*k*=10	*n*=20	*k*=1	*k*=10	*n*=20	*n*=1	*n*=1	*k*=1	*k*=1	*k*=1	*k*=1
Base	**CodeShell-7B**	9.68	16.83	45	6.82	20.11	33	4.12	7.29	49	3.59	8.58	46	47.62	43	30.43	34.30	24.63	19.78
	CodeLlama-7B	15.06	25.00	65	10.79	29.72	48	4.32	10.96	61	3.66	7.63	40	46.54	25	29.20	29.98	31.26	24.08
	CodeLlama-13B	21.91	34.85	90	12.61	38.17	67	6.14	13.83	71	4.50	8.39	44	48.60	32	32.23	35.07	35.02	23.73
	CodeLlama-34B	22.82	36.54	96	14.57	32.07	52	**7.34**	14.16	73	5.01	8.34	44	48.16	27	40.19	45.11	33.00	27.23
	StarCoderBase-15B	15.32	26.82	75	12.14	31.80	52	6.32	12.58	64	6.56	12.54	66	50.32	0	28.53	30.35	30.58	21.34
Specific Language Base	**CodeLlama-Python-7B**	10.69	21.70	59	12.00	29.06	48	4.08	8.73	46	3.23	6.96	40	49.46	12	29.15	40.48	26.41	24.56
	CodeLlama-Python-13B	14.43	26.93	72	13.00	31.68	54	3.08	7.50	41	3.72	7.07	39	47.41	37	33.56	42.89	28.11	26.23
	CodeLlama-Python-34B	14.31	26.23	72	11.71	28.24	48	5.32	11.61	63	5.54	9.27	46	49.24	15	39.46	53.29	29.17	24.73
	StarCoder-Python-15B	14.39	25.97	69	7.75	26.02	44	6.82	13.34	68	8.37	14.42	74	50.54	1	30.22	33.57	29.26	21.46
	StarCoder-Java-15B	18.82	30.28	77	10.43	32.05	56	5.75	11.11	59	6.27	10.82	57	49.14	3	30.62	27.07	31.84	14.89
Instruction Tuned	**CodeShell-Chat-7B**	7.79	15.66	43	2.82	10.97	18	3.26	7.42	38	3.79	13.99	94	50.00	0	23.57	29.66	21.22	9.93
	CodeLlama-Instruct-7B	13.38	24.91	65	3.79	15.73	29	4.84	9.60	47	3.29	14.81	93	48.92	20	28.77	45.65	21.13	10.17
	CodeLlama-instruct-13B	13.28	24.03	62	6.14	15.16	24	5.16	11.20	57	4.09	15.72	100	44.38	140	33.99	50.60	21.47	10.08
	CodeLlama-Instruct-34B	1.89	3.77	11	1.11	4.77	10	4.29	10.06	54	4.74	14.86	88	49.68	2	41.53	50.79	23.08	10.80
	WizardCoder-Python-7B	8.00	20.12	57	4.86	14.51	24	3.25	8.20	45	4.61	15.60	94	50.54	0	/	55.50	17.23	13.63
	WizardCoder-Python-13B	12.44	24.66	65	5.21	18.25	33	4.98	11.22	61	4.69	16.51	100	47.62	6	41.77	62.19	20.23	14.91
	WizardCoder-Python-34B	15.88	27.22	72	6.18	17.12	27	4.79	11.18	58	6.54	18.23	105	50.76	0	44.94	70.73	22.02	13.69
	WizardCoder-15B	14.41	23.70	64	5.00	18.84	35	3.10	11.62	67	3.89	15.94	101	33.15	308	35.77	58.12	20.67	8.36
Close Source	**Claude-1**	21.55	29.11	74	9.71	17.77	28	1.20	6.13	34	5.70	16.56	95	47.95	0	/	/	/	/
	GPT-3.5-Turbo	23.37	39.67	102	12.18	35.65	59	6.52	13.39	71	9.31	28.76	166	46.00	78	/	48.10	/	/
	GPT-4	**30.52**	**42.94**	**110**	**24.18**	**45.72**	**72**	6.66	**15.76**	**83**	**18.76**	**38.29**	**203**	**52.16**	0	/	67.00	/	/

in Humaneval. This discrepancy stems from its training corpus, which is biased towards simplistic code samples and relies on significantly fewer resources. Such results underscore the value of challenging benchmarks such as CoderUJB. Therefore, we conclude that CoderUJB provides much more challenging programming questions than HumanEval and CoderEval. Also, the other programming tasks are more complicated than the functional code generation task that previous studies [7, 10, 37, 59] mainly focused on because the requirements of the other tasks are more abstract, requiring a deeper understanding and the ability to address more complex and varied programming situations [21, 22, 56, 62].

In addition, It is also important to note that none of the current LLMs could achieve acceptable results in defect detection tasks, highlighting the formidable challenge of this task. Specifically, even GPT-4 achieves only a 52.16% accuracy rate, which is marginally better than random guessing. Previous studies [11, 15] have also found that models like ChatGPT perform inadequately when detecting common weaknesses enumerated (CWE) vulnerabilities. We believe one critical issue is that most of the code defects within CoderUJB are complex logic errors (i.e., errors producing unintended behaviors) [46] rather than syntax or API misused errors. To pinpoint such logic errors in code, a model would need an extensive grasp of the entire project, which is also challenging for experienced developers. Further complicating matters is that defect detection is a classification task [30]. This substantially differs from the mainstream pre-training tasks, i.e., autoregressive generation [26, 41]. Thus, the currently employed decoder-only autoregressive LLM suffers from inherent disadvantages when dealing with classification tasks, and such a conclusion can also be found in other natural language classification tasks [12, 48]. Given the significant challenge that defect detection poses to current LLMs, this study will not analyze the results of this task in depth. Instead, we calls for researchers to concentrate their efforts on improving the defect detection capabilities of LLMs.

Conclusion 2: In basic question-and-answer or completing scenarios, current LLMs have not achieved satisfactory results in CoderUJB representing real programming challenges, especially in the defect detection task, where all LLMs are almost randomly guessing.

Next, we compare the results of open-source LLMs and closed-source commercial LLMs to quantify the gap between the two types of models. It can be observed that the performance comparison results of the two types of LLMs differ under different programming tasks. In the area of functional code generation, the top-tier open-source LLMs (i.e., CodeLlama-34B) manage to match the performance of the well-performing closed-source model (i.e., GPT-3.5-Turbo [1]). On the two test generation tasks, top open-source LLMs (i.e., CodeLlama-34B) even surpass the GPT-3.5-Turbo. We believe that GPT-3.5-Turbo's performance may have been affected by instruction fine-tuning, which we will investigate further in Section 4.3.4. When it comes to automated program repair tasks, there's still a noticeable performance gap between the best open-source LLMs and excellent closed-source generic LLMs. The pass-all@k metrics for GPT-3.5-Turbo are 9.31 and 28.76, outperforming the corresponding metrics (6.54 and 18.32) for the top open-source LLM, WizardCoder-Python-34B.

Conclusion 3: Advance open-source LLMs have made significant progress, achieving similar or even better performance than the excellent closed-source model GPT-3.5-Turbo on the functional code generation task and the two test generation tasks, but perform poorly on the automated program repair. Meanwhile, GPT-4 surpasses all other LLMs substantially, suggesting that scaling remains a powerful tool for enhancing model performance.

4.3.3 RQ3: How Does Continued Pre-Training of Specific Programming Language (PL) Data Affect the Performance of Code LLMs Under CoderUJB. To address this research question, we look at how the Base LLMs listed in Table 4 perform compared to Specific-PL-Base LLMs under CoderUJB. In order to show the comparison results of the two classes of LLMs more intuitively, we have highlighted the results where Specific-PL-Base LLMs fall short of their Base counterparts in red and vice versa in green. We can find that the effects of specific programming language (PL) continued pre-training can vary greatly depending on the specific task.

For CoderUJB-FCG, HumanEval, and CoderEval, all of which are functional code generation tasks. Specific PL training can boost the performance of corresponding PL tasks and hinder other PL tasks. For example, CodeLlama-Python and StarCoder-Python generally perform better in HumanEval-Py and CoderEval-Py but worse in CoderEval-Java and CoderUJB-FCG (Java tasks). On the other hand, StarCoder-Java performs better in Java tasks but worse in Python tasks. This phenomenon is consistent with the consensus of researchers, i.e., in-domain training enhances the performance of in-domain tasks while potentially hindering the performance of tasks outside the domain [33, 35, 67]. However, things get different when it comes to more challenging tasks, i.e., CoderUJB-ITG and APR. The performance influence due to Specific PL training is random and less substantial in the case of those tasks when compared with functional code generation tasks. For example, we can observe two counter-instances from CoderUJB-ITG (Java tasks), i.e., StarCoder-Python getting better results in ITG after Python training while StarCoder-Java getting worse in ITG after Java training. And we can find more counter-instances in CoderUJB-APR.

We attribute this to the fact that test generation and automatic program repair tasks are more different from the pre-training task compared with functional code generation. In other words, When the downstream task is more similar to the pre-training task, such as in the case of functional code generation, the performance boost or decline is more predictable and substantial. On the other hand, when the downstream task is substantially different from the pre-training task, such as in the case of automated program repair, the effect of specific PL training tends to be unpredictable.

Furthermore, the varied outcomes across different tasks emphasize CoderUJB's value as a comprehensive evaluation benchmark that incorporates a range of programming challenges.

Conclusion 4: The impact of specific PL training might relate to how much the downstream task differs from the pre-training task. The more similar the task (e.g., functional code generation), the more predictable and substantial the performance impact (i.e., boost the performance of corresponding PL tasks and hinder other PL tasks). Conversely, if the task differs more substantially (e.g., automated program repair), the effect due to specific PL training tends to be unpredictable.

4.3.4 RQ4: How Does Instruction Fine-Tuning Influence the Performance of Code LLMs in CoderUJB. Finally, we assessed the performance of Instruction-Tuned LLMs compared to their original counterparts, as shown in Table 4. For a more intuitive comparison, we have highlighted the results where Instruction-Tuned LLMs fall short of their counterparts base models in red and vice versa in green. The results indicated that instruction tuning can yield vastly different performance impacts across diverse programming tasks.

Table 5: ***pass-syntax@k*=1 and *pass-compile@k*=1 (denoted as *p-s*@1 and *p-c*@1) results for CoderUJB.**

Model-Type	Model-ID	FCG		CTG		ITG	
		p-s@1	*p-c*@1	*p-s*@1	*p-c*@1	*p-s*@1	*p-c*@1
Base	**CodeLlama-7B**	92.63	69.58	70.64	36.71	70.14	42.36
	CodeLlama-13B	94.14	63.49	70.54	37.89	82.29	55.43
	CodeLlama-34B	96.16	61.58	77.11	39.32	87.86	51.87
Instruction Tuned	**CodeLlama-Instruct-7B**	90.53	40.65	88.64	18.04	89.39	33.34
	CodeLlama-instruct-13B	95.99	40.76	80.61	21.57	89.29	32.06
	CodeLlama-Instruct-34B	11.30	6.43	11.54	3.89	90.51	34.28

Specifically, most Instruction-Tuned LLMs struggled to outperform their base models when it came to functional code generation and test generation tasks. For instance, CodeLlama-Instruct-13B scored lower in the *pass-all@k* metrics during the function code generation task, with a 39.39% (*k*=1) and 31.05% (*k*=10) drop respectively compared to its base model (CodeLlama-13B). This performance drop was consistent among most open-source Instruction-Tuned LLMs. Such a result was different with HumanEval where instruction fine-tuning largely increased its performance. We believe that the simplicity of HumanEval's single-function generation tasks does not reflect the complexity of real-world development scenarios, a gap that CoderUJB addresses with its features. This distinction is why instruction fine-tuning has varying impacts between the two, and underscores the value of CoderUJB as a practical programming evaluation benchmark.

To investigate the cause of performance degradation, we further show the *pass-syntax@k*=1 and *pass-compile@k*=1 metrics for the three code generation tasks in Table 5. Interestingly, we found that the syntactical correctness in the code generated by the Instruction-Tuned LLMs, e.g., CodeLlama-Instruct-7B and 13B, was similar to their corresponding base models. Therefore, the decline in their performance is likely due to the lower quality of the code solutions they generate (e.g., lower *pass-compile@k* and *pass-all@k* scores) and not because of rejected answers or answers that cannot be parsed. Meanwhile, CodeLlama-Instruct-34B exhibited a noteworthy drop in *pass-syntax@k*=1 and *pass-all@k* during FCG and CTG. This decline is attributed to mode collapse [25], characterized by the generation of identical solutions for all prompts, and it occurs only in one model, leading us to disregard these results as invalid in our analysis. Thus, we believe that a possible reason is the high similarities between pre-training tasks, functional code, and test generation tasks, so that base LLMs can accomplish these tasks without aligning with upstream and downstream tasks [20, 42, 47]. Therefore, they are able to leverage the full capabilities of the base models. On the contrary, instruction tuning, with its varied form, might result in disturbances when using such diverse LLMs for direct code generation tasks.

Conversely, for the automated program repair task, instruction fine-tuning actually enhanced model performance. This trend applied to most instruction fine-tuned LLMs. We believe this is due to the significantly different format of the automated program repair task compared to the pre-training task. The differences in upstream and downstream tasks lead to poor performance when directly

[1]The results adopt from self-report value and Humaneval leader-board [8]
[2]The results were obtained through our own execution of the official evaluation scripts.

applying Base LLMs. However, instruction tuning enhances the model's adaptability and applicability to diverse tasks through data in diverse task formats [55], ultimately improving the performance of LLMs for automated program repair tasks.

Conclusion 5: Instruction tuning reduces the performance of LLMs under tasks highly consistent with the pre-trained task (e.g., functional code generation and test generation tasks), while boosting the performance of tasks that starkly differ from the pre-training tasks' format (e.g., automated program repair). Lastly, we encourage further exploration and studies to uncover more effective fine-tuning strategies for LLMs.

5 IMPLICATIONS AND DISCUSSIONS

Our study reveals the following important practical guidelines for future research on LLMs of software engineering.

Program context is important. The findings from RQ1 imply that incorporating basic program context can enhance performance across various programming tasks. Consequently, we encourage researchers to investigate and devise advanced prompt designs and methods to fully harness the potential of program context.

Caution with specific programming language continued pre-training. The results of RQ3 reveal that for tasks similar to those used in pre-training, specific PL focused training generally enhances performance in the corresponding language while potentially impeding performance in others. Conversely, if the task differs more substantially from the pre-training task, the effect due to specific PL training tends to be unpredictable. Therefore, researchers should carefully balance the training between language-specific tasks and those in other PL to determine the extent and volume of data appropriate for further pre-training.

Caution with instruction fine-tuning. The results of RQ4 indicate that instruction fine-tuning reduces the performance of LLMs under tasks highly consistent with the pre-trained task. For such tasks, we suggest researchers use the original base LLMs as the foundation for their application. Conversely, for tasks that starkly differ from the pre-training tasks, instruction-tuned LLMs tend to perform better and should be considered. Lastly, we encourage further studies to uncover more effective fine-tuning strategies.

More extensive evaluations are needed. The conclusions drawn from RQ3 and RQ4 suggest that varying programming tasks can lead to disparate results when employing the same training strategy. Consequently, we advocate that researchers should assess their LLMs and techniques using a more comprehensive benchmark such as CoderUJB to obtain more reliable evaluation outcomes.

6 THREATS TO VALIDITY

Threats to Internal Validity. The threats to internal validity mainly lie in the potential bugs in our implementation. To mitigate these risks, the authors have meticulously reviewed the code and scripts. Furthermore, we have released the code, scripts, and all generated results for public scrutiny at [4].

Threats to External Validity. This threats mainly lie in the LLMs adopted in this study. To address these concerns, we have conducted an extensive literature review and believe that the selected LLMs are sufficiently representative and influential within this field.

Threats to Construct Validity. The threats to construct validity in our study primarily arise from the metrics used in our evaluations. To mitigate these threats, we initially adopted the widely accepted *pass@k* metric and verified that each coding problem was accompanied by adequate test coverage. Additionally, we utilized a range of widely-recognized metrics, specifically *accuracy*, *count@n*, and *coverage@n*, to provide a comprehensive evaluation of the models.

7 CONCLUSIONS

CoderUJB is introduced as a benchmark that advances the evaluation of large language models (LLMs) by simulating real-world software engineering tasks with executable code extracted from 17 open-source Java projects. Our study delved into the performance of LLMs, highlighting difficulties in non-functional code generation and defect detection tasks. The research revealed the delicate balance required when continuing pre-training and instruction fine-tuning, as they can inadvertently decrease performance in certain scenarios. These observations suggest that a nuanced approach to training LLMs is essential to ensure versatility and robustness across various coding tasks. In essence, CoderUJB contributes to setting more exacting benchmarks for assessing LLMs in software engineering and provides insights into the complexities of model training, guiding future research toward developing more refined and adaptable LLMs for practical coding applications.

REFERENCES

[1] 2023. ChatGPT. Website. https://openai.com/blog/chatgpt.
[2] 2023. Claude. Website. https://www.anthropic.com.
[3] 2023. Cobertura. Website. http://cobertura.github.io/cobertura.
[4] 2023. CoderUJB. Website. https://github.com/WisdomShell/ujb.
[5] 2023. GPT-4. Website. https://openai.com/gpt-4.
[6] 2023. tiobe. Website. https://www.tiobe.com/tiobe-index/.
[7] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *CoRR* abs/2108.07732 (2021). arXiv:2108.07732 https://arxiv.org/abs/2108.07732
[8] Loubna Ben Allal, Niklas Muennighoff, Logesh Kumar Umapathi, Ben Lipkin, and Leandro von Werra. 2022. A framework for the evaluation of code generation models. https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard.
[9] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. 2023. MultiPL-E: a scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering* (2023).
[10] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison 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, Joshua 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. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374
[11] Anton Cheshkov, Pavel Zadorozhny, and Rodion Levichev. 2023. Evaluation of ChatGPT Model for Vulnerability Detection. *arXiv preprint arXiv:2304.07232* (2023).
[12] 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, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4171–4186. https://doi.org/10.18653/V1/N19-1423

[13] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. ClassEval: A Manually-Crafted Benchmark for Evaluating LLMs on Class-level Code Generation. *CoRR* abs/2308.01861 (2023). https://doi.org/10.48550/ARXIV.2308.01861 arXiv:2308.01861

[14] Sarah Fakhoury, Saikat Chakraborty, Madan Musuvathi, and Shuvendu K. Lahiri. 2023. Towards Generating Functionally Correct Code Edits from Natural Language Issue Descriptions. *CoRR* abs/2304.03816 (2023). https://doi.org/10.48550/ARXIV.2304.03816 arXiv:2304.03816

[15] Michael Fu, Chakkrit Tantithamthavorn, Van Nguyen, and Trung Le. 2023. Chatgpt for vulnerability detection, classification, and repair: How far are we? *arXiv preprint arXiv:2310.09810* (2023).

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

[17] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021. Measuring Massive Multitask Language Understanding. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=d7KBjmI3GmQ

[18] Yuzhen Huang, Yuzhuo Bai, Zhihao Zhu, Junlei Zhang, Jinghan Zhang, Tangjun Su, Junteng Liu, Chuancheng Lv, Yikai Zhang, Jiayi Lei, Yao Fu, Maosong Sun, and Junxian He. 2023. C-Eval: A Multi-Level Multi-Discipline Chinese Evaluation Suite for Foundation Models. *CoRR* abs/2305.08322 (2023). https://doi.org/10.48550/ARXIV.2305.08322 arXiv:2305.08322

[19] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 1161–1173. https://doi.org/10.1109/ICSE43902.2021.00107

[20] Mandar Joshi, Danqi Chen, Yinhan Liu, Daniel S. Weld, Luke Zettlemoyer, and Omer Levy. 2020. SpanBERT: Improving Pre-training by Representing and Predicting Spans. *Trans. Assoc. Comput. Linguistics* 8 (2020), 64–77. https://doi.org/10.1162/TACL_A_00300

[21] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, Corina S. Pasareanu and Darko Marinov (Eds.). ACM, 437–440. https://doi.org/10.1145/2610384.2628055

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

[23] Mohammad Abdullah Matin Khan, M. Saiful Bari, Xuan Long Do, Weishi Wang, Md. Rizwan Parvez, and Shafiq R. Joty. 2023. xCodeEval: A Large Scale Multilingual Multitask Benchmark for Code Understanding, Generation, Translation and Retrieval. *CoRR* abs/2303.03004 (2023). https://doi.org/10.48550/ARXIV.2303.03004 arXiv:2303.03004

[24] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2022. The Stack: 3 TB of permissively licensed source code. *CoRR* abs/2211.15533 (2022). https://doi.org/10.48550/ARXIV.2211.15533 arXiv:2211.15533

[25] Youssef Kossale, Mohammed Airaj, and Aziz Darouichi. 2022. Mode Collapse in Generative Adversarial Networks: An Overview. In *2022 8th International Conference on Optimization and Applications (ICOA)*. IEEE, 1–6.

[26] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault (Eds.). Association for Computational Linguistics, 7871–7880. https://doi.org/10.18653/V1/2020.ACL-MAIN.703

[27] 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, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Moustafa-Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, 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. 2023. StarCoder: may the source be with you! *CoRR* abs/2305.06161 (2023). https://doi.org/10.48550/ARXIV.2305.06161 arXiv:2305.06161

[28] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 169–180. https://doi.org/10.1145/3293882.3330574

[29] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: A multi-lingual program repair benchmark set based on the Quixey Challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*. 55–56.

[30] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. 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. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, Joaquin Vanschoren and Sai-Kit Yeung (Eds.). https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c16a5320fa475530d9583c34fd356ef5-Abstract-round1.html

[31] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. *arXiv preprint arXiv:2306.08568* (2023).

[32] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: combining context-aware neural translation models using ensemble for program repair. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 101–114. https://doi.org/10.1145/3395363.3397369

[33] Amil Merchant, Elahe Rahimtoroghi, Ellie Pavlick, and Ian Tenney. 2020. What Happens To BERT Embeddings During Fine-tuning?. In *Proceedings of the Third BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP, BlackboxNLP@EMNLP 2020, Online, November 2020*, Afra Alishahi, Yonatan Belinkov, Grzegorz Chrupala, Dieuwke Hupkes, Yuval Pinter, and Hassan Sajjad (Eds.). Association for Computational Linguistics, 33–44. https://doi.org/10.18653/V1/2020.BLACKBOXNLP-1.4

[34] Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. 2022. Rethinking the Role of Demonstrations: What Makes In-Context Learning Work?. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (Eds.). Association for Computational Linguistics, 11048–11064. https://doi.org/10.18653/V1/2022.EMNLP-MAIN.759

[35] Marius Mosbach, Maksym Andriushchenko, and Dietrich Klakow. 2021. On the Stability of Fine-tuning BERT: Misconceptions, Explanations, and Strong Baselines. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=nzpLWnVAyah

[36] Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. 2023. Lever: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning*. PMLR, 26106–26128.

[37] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/pdf?id=iaYcJKpY2B_

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

[39] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. 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 *NeurIPS*. http://papers.nips.cc/paper_files/paper/2022/hash/b1efde53be364a73914f58805a001731-Abstract-Conference.html

[40] Chen Qian, Xin Cong, Cheng Yang, Weize Chen, Yusheng Su, Juyuan Xu, Zhiyuan Liu, and Maosong Sun. 2023. Communicative agents for software development. *arXiv preprint arXiv:2307.07924* (2023).

[41] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. (2018).

[42] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *J. Mach. Learn. Res.* 21 (2020), 140:1–140:67. http://jmlr.org/papers/v21/20-074.html

[43] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. *CoRR* abs/2009.10297 (2020). arXiv:2009.10297 https://arxiv.org/abs/2009.10297

[44] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, 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. 2023. Code Llama: Open Foundation Models for Code. *CoRR* abs/2308.12950 (2023). https://doi.org/10.48550/ARXIV.2308.12950 arXiv:2308.12950

[45] Bo Shen, Jiaxin Zhang, Taihong Chen, Daoguang Zan, Bing Geng, An Fu, Muhan Zeng, Ailun Yu, Jichuan Ji, Jingyang Zhao, et al. 2023. Pangu-coder2: Boosting large language models for code with ranking feedback. *arXiv preprint arXiv:2307.14936* (2023).

[46] Dowon Song, Myungho Lee, and Hakjoo Oh. 2019. Automatic and scalable detection of logical errors in functional programming assignments. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 188:1–188:30. https://doi.org/10.1145/3360614

[47] Yixuan Su, Lei Shu, Elman Mansimov, Arshit Gupta, Deng Cai, Yi-An Lai, and Yi Zhang. 2022. Multi-Task Pre-Training for Plug-and-Play Task-Oriented Dialogue System. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). Association for Computational Linguistics, 4661–4676. https://doi.org/10.18653/V1/2022.ACL-LONG.319

[48] Xiaofei Sun, Xiaoya Li, Jiwei Li, Fei Wu, Shangwei Guo, Tianwei Zhang, and Guoyin Wang. 2023. Text Classification via Large Language Models. *CoRR* abs/2305.08377 (2023). https://doi.org/10.48550/ARXIV.2305.08377 arXiv:2305.08377

[49] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *CoRR* abs/2307.09288 (2023). https://doi.org/10.48550/ARXIV.2307.09288 arXiv:2307.09288

[50] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[51] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2023. Software Testing with Large Language Model: Survey, Landscape, and Vision. *CoRR* abs/2307.07221 (2023). https://doi.org/10.48550/ARXIV.2307.07221 arXiv:2307.07221

[52] Yidong Wang, Hao Chen, Yue Fan, Wang Sun, Ran Tao, Wenxin Hou, Renjie Wang, Linyi Yang, Zhi Zhou, Lan-Zhe Guo, et al. 2022. Usb: A unified semi-supervised learning benchmark for classification. *Advances in Neural Information Processing Systems* 35 (2022), 3938–3961.

[53] Yidong Wang, Zhuohao Yu, Jindong Wang, Qiang Heng, Hao Chen, Wei Ye, Rui Xie, Xing Xie, and Shikun Zhang. 2024. Exploring vision-language models for imbalanced learning. *International Journal of Computer Vision* 132, 1 (2024), 224–237.

[54] Yidong Wang, Zhuohao Yu, Zhengran Zeng, Linyi Yang, Cunxiang Wang, Hao Chen, Chaoya Jiang, Rui Xie, Jindong Wang, Xing Xie, et al. 2023. Pandalm: An automatic evaluation benchmark for llm instruction tuning optimization. *arXiv preprint arXiv:2306.05087* (2023).

[55] Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. 2022. Finetuned Language Models are Zero-Shot Learners. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022.* OpenReview.net. https://openreview.net/forum?id=gEZrGCozdqR

[56] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-trained Language Models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023.* IEEE, 1482–1494. https://doi.org/10.1109/ICSE48619.2023.00129

[57] Rui Xie, Zhengran Zeng, Zhuohao Yu, Chang Gao, Shikun Zhang, and Wei Ye. 2024. CodeShell Technical Report. arXiv:2403.15747 [cs.SE]

[58] Zhou Yang, Zhipeng Zhao, Chenyu Wang, Jieke Shi, Dongsun Kim, DongGyun Han, and David Lo. 2023. What do code models memorize? an empirical study on large language models of code. *arXiv preprint arXiv:2308.09932* (2023).

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

[60] Zhuohao Yu, Chang Gao, Wenjin Yao, Yidong Wang, Wei Ye, Jindong Wang, Xing Xie, Yue Zhang, and Shikun Zhang. 2024. KIEval: A Knowledge-grounded Interactive Evaluation Framework for Large Language Models. *arXiv preprint arXiv:2402.15043* (2024).

[61] Zhiqiang Yuan, Junwei Liu, Qiancheng Zi, Mingwei Liu, Xin Peng, and Yiling Lou. 2023. Evaluating instruction-tuned large language models on code comprehension and generation. *arXiv preprint arXiv:2308.01240* (2023).

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

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

[64] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. 2023. A Survey of Large Language Models. *arXiv preprint arXiv:2303.18223* (2023). http://arxiv.org/abs/2303.18223

[65] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. 2023. 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.* 5673–5684.

[66] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 10197–10207. https://proceedings.neurips.cc/paper/2019/hash/49265d2447bc3bbfe9e76306ce40a31f-Abstract.html

[67] Yichu Zhou and Vivek Srikumar. 2022. A Closer Look at How Fine-tuning Changes BERT. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). Association for Computational Linguistics, 1046–1061. https://doi.org/10.18653/V1/2022.ACL-LONG.75

Received 15-DEC-2023; accepted 2024-03-02