CodeScope: An Execution-based Multilingual Multitask Multidimensional Benchmark for Evaluating LLMs on Code Understanding and Generation

Anonymous ACL submission

Abstract

Large Language Models (LLMs) have demonstrated remarkable performance on assisting humans in programming and facilitating programming automation. However, existing benchmarks for evaluating the code understanding and generation capacities of LLMs suffer from severe limitations. First, most benchmarks are insufficient as they focus on a narrow range of popular programming languages and specific tasks. Second, most benchmarks fail to consider the actual executability and the consistency of execution results of the generated code. To bridge these gaps between existing benchmarks and expectations from practical applications, we introduce **CodeScope**, an execution-based, multilingual, multitask, multidimensional evaluation benchmark for comprehensively measuring LLM capabilities on coding tasks. CodeScope covers **43 programming languages** and **eight coding tasks**. It evaluates the coding performance of LLMs from three dimensions (perspectives): **length**, **difficulty**, and **efficiency**. To facilitate execution-based evaluations of code generation, we develop **MultiCodeEngine**, an automated code execution engine that supports 14 programming languages. Finally, we systematically evaluate and analyze eight mainstream LLMs and demonstrate the superior breadth and challenges of CodeScope for evaluating LLMs on code understanding and generation tasks compared to other benchmarks[1].

1 Introduction

Driven by advances in deep learning and NLP, LLMs have demonstrated outstanding proficiency in various generation and understanding tasks (OpenAI, 2023; Anil et al., 2023). However, existing benchmarks (Hendrycks et al., 2020; Zhong et al., 2023; Zheng et al., 2023a) for evaluating LLMs mainly focus on NLP tasks, such as common sense reasoning, academic examination, and authenticity verification. Existing evaluation methods are significantly insufficient in terms of evaluating completeness and comprehensiveness for code understanding and generation capabilities of LLMs. Firstly, many code LLMs, such as CodeT5+ (Wang et al., 2023b), WizardCoder (Luo et al., 2023), and Code LLaMA (Rozière et al., 2023), employ their own specific single-task evaluation datasets, making it infeasible to comprehensively compare the performance of various LLMs on code understanding and generation tasks on a unified standard.

Category	Dimension	Task	#Lang.	#Samples	Length
Understanding	Length	Code Summarization	43	4,838	385
		Code Smell	2	200	650
		Code Review	9	900	857
		Automated Testing	4	400	251
Generation	Difficulty	Program Synthesis	14	803	538
		Code Translation	14	5,382	513
		Code Repair	14	746	446
	Efficiency	Code Optimization	4	121	444

Table 1: Summary of our **CodeScope**. We report the number of language (#Lang.) and samples (#Samples) and the average number of tokens per sample (Length) for test sets of each task. Token counts are based on OpenAI's tiktoken tokenizer (https://github.com/openai/tiktoken).

Secondly, existing datasets mostly evaluate LLMs on code tasks (Chen et al., 2021; Austin et al., 2021) for a narrow range of popular programming languages, with a focus on Python and single program synthesis tasks. However, software development often involves multiple programming languages, each following different programming paradigms such as object-oriented, functional, and procedural. Evaluating LLMs within a multilingual framework can reveal their ability to generalize across various languages and paradigms.

Thirdly, most studies (e.g., widely used benchmarks CodeXGLUE (Lu et al., 2021) and XLCoST (Zhu et al., 2022)) rely on matching-based evaluation metrics, such as BLEU or CodeBLEU,

[1]The CodeScope benchmark and datasets are publicly available at `placeholder`.

to measure the quality of generated code. However, these metrics may not reflect the practical applicability of the code, as they only compare the surface form similarity between the generated code and the reference code (Yan et al., 2023). The ultimate goal of code generation is to produce code that can execute correctly and accomplish specific tasks, regardless of the implementation details. Therefore, execution-based metrics, which evaluate the functionality and correctness of the generated code by running it on test cases or comparing its output with the expected output, are more reliable and informative.

To address these limitations, we propose **CodeScope**, a benchmark that evaluates the coding proficiency of LLMs using execution-based metrics in a *multilingual* and *multitask* setting. CodeScope consists of eight tasks for code understanding and generation, covering 43 programming languages with an average of 13 languages per task. The task descriptions are summarized in Table 1. We also conduct comprehensive evaluations of LLMs across three dimensions (that is, *multidimensional*): **Length**, **Difficulty**, and **Efficiency**. Length measures the ability to process code of different lengths, Difficulty evaluates proficiency in solving increasingly complex programming challenges, and Efficiency examines the execution speed and resource consumption of the code generated by LLMs for a specific Code Optimization task.

To support CodeScope, we develop a Multilingual Code Execution Engine, **MultiCodeEngine**, which extends the ExecEval engine (Khan et al., 2023) to accommodate 14 programming languages for code generation tasks. We also establish eight strong baselines for each task to facilitate comprehensive comparisons of coding capabilities of LLMs. We expect these explorations will provide a deep understanding of the strengths and limitations of LLMs on code understanding and generation tasks and provide valuable guidance for future research directions. Our contributions can be summarized as follows:

- **CodeScope benchmark**: We built the first-ever comprehensive benchmark for evaluating LLMs on code understanding and generation tasks, **CodeScope**, which covers the largest number of programming languages (43 in total) and comprises the most comprehensive spectrum of diverse code understanding and generation tasks (eight tasks in total) to date. This benchmark evaluates the actual execution of the generated code, facilitated by MultiCodeEngine, a multilingual code execution engine supporting 14 programming languages.
- **Multidimensional fine-grained evaluation**: We comprehensively evaluate the performance of LLMs on **eight tasks** from **three dimensions**, namely, **length** (i.e., length of code required to solve the problem); **difficulty** (i.e., complexity of programming problems); and **efficiency** (i.e., execution efficiency of generated code).
- **Comprehensive evaluations and in-depth analyses**: We evaluate and compare the coding capabilities of eight mainstream LLMs and establish strong baselines for each task. We conduct comprehensive validations and analyses of the utility of the CodeScope benchmark.

2 Related Work

Many existing benchmarks for code understanding and generation tasks do not use execution-based evaluations. For example, CodeXGLUE (Lu et al., 2021) and XLCoST (Zhu et al., 2022) only use matching-based metrics, such as BLEU or CodeBLEU, which compare the surface form similarity between the generated code and the reference code. However, these metrics may not capture the practical applicability of the code, as they can be misled by syntactically correct but semantically incorrect code, or by different implementations of the same functionality. Previous research has shown that code lexical similarity and execution correctness are weakly correlated (Chen et al., 2021; Austin et al., 2021; Ren et al., 2020). A recent benchmark, XCodeEval (Khan et al., 2023), uses execution-based metrics in a multilingual and multitask setting, but some of its tasks are not relevant for LLMs, such as code retrieval, which requires a large and reliable code knowledge base that is not yet available for LLMs. Furthermore, we found several flaws in the XCodeEval dataset, such as the inclusion of Russian language data, which biases the natural language understanding of the instructions; inconsistencies between test cases and actual execution outputs; and the presence of invalid "cheat codes" that users have submitted to the website.

In addition, most of the related works (Hendrycks et al., 2021; Lai et al., 2023; Huang et al., 2022; Nijkamp et al., 2023; Chandel et al., 2022) evaluate model performance on program synthesis tasks with Python as the target language. Among the existing datasets for program synthesis, HumanEval (Chen et al., 2021) is the most popular, with 164 problems and an average of 6.7 unit tests per problem. MBPP (Austin et al., 2021) contains 974 entry-level programming tasks, while MathQA (Austin et al., 2021) includes 23,914 more advanced programming problems. APPS (Hendrycks et al., 2021) is designed to pose more challenging programming problems. However, the coverage of programming languages in program synthesis tasks is still limited. Some recent studies (Yu et al., 2023; Li et al., 2022a) have attempted to expand the range of programming languages in program synthesis task, but they only cover a few languages. MBXP (Athiwaratkun et al., 2023) is a dataset that covers ten programming languages, generated by a scalable transformation framework. HumanEval-X (Zheng et al., 2023b) is another dataset that covers five programming languages, created by human translation. Moreover, some recent research (Yu et al., 2023) has pointed out the limitations of HumanEval in evaluating the contextual appropriateness of the generated code. We provide a more detailed discussion of other code evaluation benchmarks in Section A.1 of the appendix.

3 The CodeScope Benchmark

CodeScope evaluates the performance of LLMs on both code understanding and generation tasks. More details on dataset construction for each task are in Appendix A.2.1 to A.2.8.

3.1 Code Understanding

The code understanding tasks aim to evaluate the LLMs' ability to comprehend and analyze code. The tasks include **code summarization**, which requires the model to concisely summarize the core functionality and intent of the code; **code smell**, which requires the model to detect potential programming issues and poor practices *in snippets within the input code*; **code review**, which requires the model to evaluate the *overall* quality, style, and errors of the code; and **automated testing**, which requires the model to understand the programming logic, data flow, and execution process of the code. Figure 1 shows the main workflow for evaluating LLMs on the four code understanding tasks.

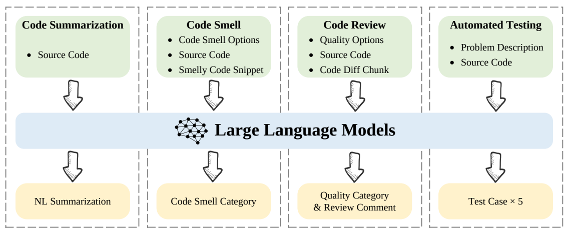

Figure 1: The task definition of the four code understanding tasks.

3.1.1 Code Summarization

Task Definition Code summarization aims to summarize the functionality and intent of source code into concise natural language descriptions (PL-to-NL), assisting developers in quickly grasping the functionality and behavior of the code. Code summarization requires the model to not only accurately recognize the structure of the code but also understand how its components collaboratively achieve specific functions. The input of this task is a snippet of source code and the output is its functional description in natural language.

Data Characteristics Since each programming language has its own distinct syntax, semantics, and usage patterns, evaluating code summarization capabilities based solely on a handful of mainstream programming languages is insufficient. Hence, we collect code summarization data for the 43 most popular programming languages from the Rosetta Code website[2]. To the best of our knowledge, our code summarization dataset covers the largest number of programming languages.

Evaluation Metrics We employ four commonly used metrics, BLEU, METEOR, ROUGE, and BERTScore, for evaluating code summarization.

3.1.2 Code Smell

Task Definition Code smells are indicators of bad design choices that degrade the quality of the software system, without necessarily affecting its functionality or correctness. However, these code smells can result in lower system performance and higher likelihood of future errors. To identify code smells accurately, LLMs need to analyze and understand the source code from both global and local perspectives. The task input consists of a smelly code snippet, the source code where it belongs, and five possible code smell categories. The task output is the correct code smell category for the snippet.

[2]https://rosettacode.org/wiki/Rosetta_Code

Data Characteristics We select a subset of samples from the Java and C# datasets published by Madeyski and Lewowski (2023); Slivka et al. (2023), covering three class-level and two method-level code smell categories.

Evaluation Metrics We adopt common classification evaluation metrics for the five-class classification task of code smells, including accuracy, precision, recall, and weighted F1 score.

3.1.3 Code Review

Task Definition Code review is the process of systematically examining source code written by other developers to find and fix potential errors and to ensure that the code follows the team's coding standards. Code review tasks can measure the understanding and analysis skills of LLMs by asking them to judge and comment on the code. We use two reviewer-perspective tasks from Li et al. (2022b) to evaluate the code review skills of large language models: quality estimation and code review generation. The quality estimation task is a binary classification task that predicts whether code changes need further comments or suggestions. The input is the code changes, and the output is either *comments required* or *no comments required*. The code review generation task is a sequence generation task that generates comments or suggestions for code changes that need improvement. The input is the same code changes, and the output is the generated natural language comment.

Data Characteristics We use the code quality estimation dataset released by Li et al. (2022b), which includes real-world code changes, quality estimation, and review comment data in Github, covering nine commonly used programming languages.

Evaluation Metrics Quality estimation use accuracy, precision, recall and weighted F1 scores as evaluation metrics. For the evaluation of comment generation, we employ the BLEU, ROUGE, and BERTScore as our evaluation metrics.

3.1.4 Automated Testing

Task Definition Automated testing refers to running test cases that are automatically generated through specific tools or scripts, aiming to quickly and comprehensively verify code functionality and performance without human intervention to ensure that it meets expected requirements. Automatically generated test cases play a key role in identifying and locating defects and errors in the code, which can effectively ensure the stability and reliability of the code. Automated testing requires the LLM to understand the core purpose of the code, identify potential boundary conditions and constraints, and grasp the flow and transformation of data during the code's execution. The task input is the problem description and the corresponding code solution, and the output is a set of test cases.

Data Characteristics We construct an automated testing dataset using samples of four programming languages Python, Java, C, and C++, which we crawl from Codeforces[3], a popular online algorithm competition platform.

Evaluation Metrics We use three metrics to measure the quality of test cases generated by LLMs: pass rate, line coverage, and branch coverage. The pass rate is the percentage of test cases that pass the test, which means they meet the format requirements, execute correctly, and produce the expected output. Line coverage is the percentage of code lines that are covered by test cases out of the total number of code lines. Branch coverage is the percentage of branches that are executed by test cases out of all possible branches in the code.

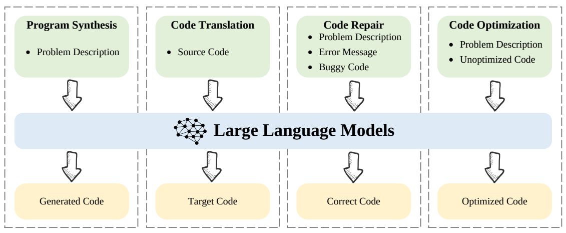

Figure 2: The task definition of the four code generation tasks.

3.2 Code Generation

Compared to code understanding, code generation tasks require LLMs to produce target code that meets various requirements. The tasks are: **Program synthesis** (*Correctness*), which evaluates the ability of LLMs to generate correct code according to the given NL-description; **Code translation** (*Compatibility*), which examines whether LLMs can maintain functional consistency when translating between different programming languages; **Code repair** (*Maintainability*), which focuses on LLMs' ability to detect and fix errors automatically; and **Code optimization** (*Efficiency*), which evaluates LLMs' capability to improve the performance and resource consumption of the code. Figure 2

[3]https://codeforces.com

shows the main workflow for evaluating LLMs on the four code generation tasks.

3.2.1 Program Synthesis (NL-to-PL)

Task Definition The objective of program synthesis is to generate expected code solutions based on the natural language description of the task. Program synthesis not only requires LLMs to have strong logical reasoning and problem-solving abilities, but also examines the ability of LLMs to accurately express logical structures into concrete code at a deeper level. The input is a programming scenario described in natural language, including sample inputs and outputs of the problem, while the expected output is code that can solve the corresponding problem.

Data Characteristics Given that LLMs should be able to generate code in various programming languages, we have designed the most diverse set of execution-based program synthesis tasks so far, covering 14 programming languages with different levels of resources. Unlike existing benchmarks (Chen et al., 2021; Austin et al., 2021) that give clear and simple descriptions of programming requirements, we evaluate the LLMs' ability to solve real-world coding problems with increasing difficulty. This requires LLMs to not only understand the task description, but also to design or choose suitable programming algorithms and generate the corresponding solutions.

We construct the **Codeforces4LLM** dataset by collecting data from Codeforces. According to the TIOBE Programming Community Index[4], we collect problem descriptions and correct submissions of corresponding problems in 14 different programming languages. According to the official difficulty standards of the Codeforces platform, we set two difficulty levels for each programming language: Easy ([800, 1600)) and Hard ([1600, 2800))[5].

Evaluation Metrics We adopt the execution-based metric *Pass@k* (Chen et al., 2021) to evaluate the code generated by LLMs. To facilitate this evaluation metric, we develop a multilingual integrated execution testing environment, called **MultiCodeEngine**, which can support 47 compiler/interpreter versions across the 14 programming languages involved in code generation.

3.2.2 Code Translation (PL-to-PL)

Task Definition The objective of code translation is to convert source code from one programming language to another, promoting software compatibility across different platforms, and supporting the maintenance and modernization of early software systems. This process requires LLMs not only to achieve functional equivalence in execution-based evaluation, but also to identify dependencies and edge cases in different programming languages. Its input includes the source code of a specific language and the designation of the target programming language, and the expected output is the corresponding and functionally consistent code in the target programming language.

Data Characteristics We follow the same programming language coverage and task difficulty settings as program synthesis task. We utilize the Codeforces4LLM dataset in our program synthesis task.

Evaluation Metrics We adopt the same metrics *pass@k*, and use the MultiCodeEngine as our execution environment for both code translation and program synthesis tasks.

3.2.3 Code Repair (NL&PL-to-PL)

Task Definition The objective of code repair is to identify and correct errors or defects in source code to ensure that the code performs correctly and meets expected functional requirements. Code repair integrates a series of complex and diverse challenges, including fine-grained code understanding, problem diagnosis across NL and PL, and formulating effective repair strategies. Its input includes the error code snippet, the corresponding problem description, and the error message returned by the compiler/interpreter, while the expected output is the corrected code that solves the corresponding problem.

Data Characteristics We follow the same programming language coverage and task difficulty settings as program synthesis tasks. We expand the Codeforces4LLM dataset by collecting additional incorrect code submissions for each problem and execute them in the MultiCodeEngine to obtain code error information.

Evaluation Metrics Given that the code repair task measures the LLMs' skill in finding and fixing specific errors or bugs in the code, we use the *Debugging Success Rate@K* (DSR@K) metric (Yan et al., 2023) to evaluate the execution-based code repair capabilities of LLMs. The DSR@K met-

[4]TIOBE Programming Community Index is a metric of the popularity of programming languages.

[5]Among them, a difficulty rating of 800 represents the lowest level of challenge on the website. As this number increases, it indicates a corresponding rise in both the complexity and difficulty of the problems to be solved.

Code Summarization					
Model	**Short**	**Medium**	**Long**	**Avg.**	**SD**
GPT-4	33.78	33.27	33.88	**33.66**	0.33
GPT-3.5	33.21	32.87	33.51	33.14	**0.32**
Vicuna	32.12	32.21	31.62	32.06	**0.32**
WizardCoder	32.85	32.05	29.01	31.99	2.03
Code LLaMA	32.39	31.36	28.59	31.52	1.97
LLaMA 2	32.03	31.25	29.34	31.40	1.38
StarCoder	31.63	30.69	30.08	31.18	0.78
PaLM 2	31.83	29.95	24.20	30.27	3.98

Code Smell					
Model	**Short**	**Medium**	**Long**	**Avg.**	**SD**
WizardCoder	45.09	48.29	53.03	**48.80**	**3.99**
LLaMA 2	41.13	31.77	49.28	40.73	8.76
Vicuna	38.94	30.66	39.54	36.38	4.96
GPT-4	30.44	40.02	37.60	36.02	4.98
PaLM 2	28.48	41.61	36.14	35.41	6.60
GPT-3.5	29.12	38.13	37.55	34.93	5.04
Code LLaMA	34.78	40.79	24.10	33.22	8.45
StarCoder	28.75	19.79	14.13	20.89	7.37

Code Review					
Model	**Short**	**Medium**	**Long**	**Avg.**	**SD**
Code LLaMA	39.34	44.70	43.66	**42.57**	2.84
GPT-4	44.08	39.93	41.69	41.90	2.08
LLaMA 2	45.74	40.05	39.14	41.64	3.58
PaLM 2	41.56	42.13	39.79	41.16	**1.22**
Vicuna	43.92	38.70	40.43	41.02	2.66
GPT-3.5	45.75	37.88	34.56	39.40	5.75
WizardCoder	32.68	41.05	43.36	39.03	5.62
StarCoder	45.34	39.02	32.20	38.85	6.57

Automated Testing					
Model	**Short**	**Medium**	**Long**	**Avg.**	**SD**
GPT-3.5	87.49	86.37	80.91	**84.92**	3.52
PaLM 2	84.52	81.97	80.38	82.29	2.09
LLaMA 2	83.46	80.48	80.27	81.40	1.78
Code LLaMA	82.65	79.34	80.27	80.75	**1.71**
WizardCoder	82.25	82.13	77.87	80.75	2.49
StarCoder	78.70	80.77	72.96	77.48	4.05
GPT-4	80.80	75.03	75.33	77.05	3.25
Vicuna	75.19	74.85	79.15	76.40	2.39

Length		
Model	**Overall**	**SD**
WizardCoder	50.14	3.53
LLaMA 2	48.79	3.88
GPT-3.5	48.10	3.66
PaLM 2	47.28	3.47
GPT-4	47.16	2.66
Code LLaMA	47.02	3.74
Vicuna	46.47	2.68
StarCoder	42.10	4.69

Table 2: Performance comparison of LLMs in code understanding tasks at different length levels. **Short**, **Medium**, and **Long** are the length classifications of the code. **SD** means standard deviation.

ric counts a code sample as successfully repaired if it produces the expected output after at most K rounds of debugging, when it did not do so before.

3.2.4 Code Optimization

Task Definition Code optimization is the process of improving the time or space complexity of a program without changing its intended functionality. The goal is to increase execution efficiency, which saves time and hardware resources. Efficiency optimization can be done at the compiler level, or by transforming the source code (data structures, algorithms, or language syntax). Code optimization in CodeScope focuses on improving code efficiency from the source code perspective. To the best of our knowledge, CodeScope is the first work to explore the capabilities of LLMs in code optimization. The input includes the problem description, the source code awaiting optimization, the specified programming language, and representative test case inputs and outputs. The output is the optimized code.

Data Characteristics We screen Codeforces4LLM to construct the code optimization dataset, specifically selecting 30 programming tasks in each of the four prevalent programming languages Python 3, C#, C, and C++.

Evaluation Metrics Given that code optimization measures the LLMs' skill in finding and improving inefficient code, we propose a novel metric, **Opt@K**, to quantify this skill. Opt@K assumes that a code sample that can be optimized for efficiency is successfully optimized if any of the optimized code samples has higher efficiency than the original sample in K optimization attempts. We measure the efficiency of code samples by recording their execution time and memory usage during the code execution process.

4 Multidimensional Evaluation

We present eight popular large language models along with their performance on various tasks and analyze the experimental results based on different dimensions. Additionally, we report in detail the specific information of baseline LLMs, the parameter setting of the experiment, and the hardware information used for inference in Appendix A.3.

4.1 Length

Table 2 presents the performance and stability of various LLMs in code understanding tasks across evaluations of different lengths. Detailed experimental results are provided in Tables 7 to 10 in the appendix. Additionally, case studies for each task are reported in Tables 35 to 42 in the appendix.

Performance WizardCoder demonstrates the best performance among all the tested LLMs, with an overall performance of 50.14, showing its significant advantage in understanding and processing complex code structures. This advantage is attributed to its Evol-Instruct approach, which significantly enhances the model's understanding by fine-tuning it with open-domain instructions across varying levels of difficulty and technical scopes. Notably, GPT-4 does not exhibit leading performance, mainly due to its poor performance in automated testing tasks. Our analysis of GPT-4's

Program Synthesis				Code Translation				Code Repair				Difficulty	
Model	Easy	Hard	Avg.	Model	Easy	Hard	Avg.	Model	Easy	Hard	Avg.	Model	Overall
GPT-4	58.57	12.01	**36.36**	GPT-4	40.26	22.06	**31.29**	GPT-4	43.56	14.04	**30.03**	GPT-4	**32.56**
GPT-3.5	39.29	4.96	22.91	GPT-3.5	28.50	14.03	21.37	GPT-3.5	18.56	7.60	13.54	GPT-3.5	19.27
Code LLaMA	7.14	0.26	3.86	WizardCoder	8.83	3.24	6.07	PaLM 2	7.43	7.02	7.24	WizardCoder	4.85
WizardCoder	5.95	0.26	3.24	StarCoder	5.75	1.89	3.85	Wizardcoder	4.95	5.56	5.23	PaLM 2	4.25
PaLM 2	3.81	0.78	1.99	PaLM 2	5.27	1.70	3.51	Code LLaMA	4.21	3.51	3.89	Code LLaMA	3.68
LLaMA 2	1.43	0.00	0.75	Code LLaMA	4.91	1.66	3.31	Vicuna	3.47	2.34	2.95	StarCoder	2.39
StarCoder	0.95	0.00	0.50	LLaMA 2	1.10	0.26	0.69	Starcoder	2.23	3.51	2.82	Vicuna	1.24
Vicuna	0.71	0.00	0.37	Vicuna	0.62	0.19	0.41	LLaMA 2	1.49	1.46	1.47	LLaMA 2	0.97

Table 3: Performance comparison in program synthesis, code translation, code repair at varying difficulty levels, evaluated using Pass@5, Pass@1, DSR@1 testing. **Easy** and **Hard** categories refer to the difficulty.

Model	Python		C		C++		C#		Overall
	Memory	Time	Memory	Time	Memory	Time	Memory	Time	
GPT-4	46.67	36.67	43.33	6.67	29.04	3.23	36.67	23.33	**28.20**
GPT-3.5	40.00	20.00	76.67	6.67	29.03	19.35	0.00	20.00	26.46
WizardCoder	50.00	16.67	50.00	0.00	38.71	12.90	10.00	16.67	24.37
Code LLaMA	43.33	13.33	40.00	0.00	35.48	3.22	10.00	23.33	21.09
PaLM 2	20.00	13.33	20.00	0.00	6.45	6.45	0.00	6.67	9.11
StarCoder	20.00	6.67	13.33	0.00	16.13	0.00	3.33	6.67	8.27
LLaMA 2	16.67	3.33	16.67	6.67	6.45	0.00	6.67	0.00	7.06
Vicuna	20.00	6.67	13.33	0.00	6.45	0.00	0.00	6.67	6.64

Table 4: Performance comparison of LLMs in code optimization under different efficiency perspectives, evaluated using Opt@5 testing.

experimental results finds that it struggles to generate test cases consistent with actual execution outputs, indicating that GPT-4 still has room for improvement in tracking and analyzing data flow during code execution.

Stability To measure the stability of LLMs when processing code of different lengths, we use the standard deviation of their performance. GPT-4 and Vicuna show excellent stability, with a standard deviation of only 2.66 and 2.68, respectively, which means they handle texts of various lengths consistently and stably. Interestingly, some models perform better with longer codes, which may be due to their strong contextual understanding and the abundance of long code samples in their training datasets.

4.2 Difficulty

Table 3 presents the performance of various LLMs in tasks of program synthesis, code translation, and code repair across evaluations of different difficulties. Detailed experimental results are provided in Tables 12 to 31 in the appendix, while case studies for each task are reported in Tables 43 to 46.

GPT-4 and GPT-3.5 excel in three different code generation tasks, thanks to their advanced training methods and high-quality data. GPT-3.5 performs well on easy problems, but GPT-4 outperforms it on harder ones. Setting different levels of difficulty helps to show the strengths and weaknesses of various LLMs, and shows the importance of choosing the right difficulty level when evaluating LLMs. Other LLMs lag behind GPT-4 and GPT-3.5 on both easy and hard tasks. They struggle to provide correct solutions for hard problems, which limits their usefulness in real-world programming applications. For these LLMs, it is easier to fix buggy code than to generate solutions from scratch. CodeScope is a valuable addition to the field of code generation, as it can evaluate the LLMs' ability to solve real-world programming problems more accurately. CodeScope solves the problem of HumanEval's benchmark accuracy rate being too high (94.4%) (Zhou et al., 2023), which means it is too easy.

4.3 Efficiency

As Table 4 shows, GPT-4 performs the best among various LLMs in the overall evaluation of code optimization, especially in reducing execution time. GPT-4 is not always the best in memory optimization, but it is consistent across different programming languages. WizardCoder and Code LLaMA also perform well in code optimization, compared to GPT-4 and GPT-3.5, which shows their awareness of memory usage and time efficiency during code execution.

We notice that LLMs optimize Python code the best, but C code the worst, especially in terms of execution time. This may be because C language has low-level features and strict details, such as accurate memory management and pointer operations. We also notice that most successful optimization cases are only at the syntactic level, where LLMs tend to use syntactic improvement strategies. To present our code optimization process more comprehensively, we provide case studies of code optimization in Appendix Tables 47 and 49.

Ranking	**CodeScope** (Understanding)	**CodeScope** (Generation)	**CodeScope** (Overall)	**HumanEval Pass@1**	**MBPP Pass@1**
1	WizardCoder (50.14)	GPT-4 (31.47)	GPT-4 (39.31)	GPT-4 (67.0)	GPT-4 (61.8)
2	LLaMA 2 (48.79)	GPT-3.5 (21.07)	GPT-3.5 (34.58)	WizardCoder (57.3)	Code LLaMA (57.0)
3	GPT-3.5 (48.10)	WizardCoder (9.73)	WizardCoder (29.94)	GPT-3.5 (48.1)	GPT-3.5 (52.2)
4	PaLM 2 (47.28)	Code LLaMA (8.04)	Code LLaMA (27.53)	Code LLaMA (41.5)	WizardCoder (51.8)
5	GPT-4 (47.16)	PaLM 2 (5.46)	PaLM 2 (26.37)	PaLM 2 (37.6)	PaLM 2 (50.0)
6	Code LLaMA (47.02)	StarCoder (3.86)	LLaMA 2 (25.64)	StarCoder (33.6)	LLaMA 2 (45.4)
7	Vicuna (46.47)	Vicuna (2.59)	Vicuna (24.53)	LLaMA 2 (30.5)	StarCoder (43.6)
8	StarCoder (42.10)	LLaMA 2 (2.49)	StarCoder (22.98)	Vicuna (15.2)	Vicuna (22.4)

Table 5: Comparison of results of eight baseline models on CodeScope, HumanEval and MBPP benchmarks.

5 Comparison with HumanEval and MBPP Benchmarks

Table 5 compares the performance of eight widely-used LLMs on the CodeScope, HumanEval, and MBPP benchmarks[6]. Unlike HumanEval and MBPP, which mainly focus on one aspect of evaluation, CodeScope evaluates LLMs from both code understanding and code generation perspectives, providing a more balanced and comprehensive framework.

In CodeScope (Understanding), we evaluate the LLMs' skill in interpreting and analyzing code. We calculate the average performance of each model on four code understanding tasks, and use it as their overall score in this domain, as shown in Table 5. The rankings of these LLMs in code understanding are different from their rankings in HumanEval and MBPP. For example, GPT-4, which is the best in HumanEval and MBPP, is only fifth in CodeScope (Understanding). This shows that good performance in code generation tasks does not mean good understanding of complex code.

In CodeScope (Generation), we use the same method to calculate the overall score. GPT-4 and GPT-3.5 do much better in code generation than in HumanEval and MBPP. We think this may be because of two reasons. First, CodeScope (Generation) tests the general ability of LLMs to generate code for multiple objectives and languages. Unlike HumanEval and MBPP, which only test *NL-to-PL* tasks, CodeScope tests *NL-to-PL*, *PL-to-PL*, and *NL&PL-to-PL* tasks, and evaluates the correctness, quality, and efficiency of the generated code, as well as the adaptability of LLMs to different languages. Second, CodeScope (Generation) has more complex and diverse problems, with different levels of difficulty. In contrast, HumanEval and MBPP have simple and predefined problems. For example, the average number of tokens in solutions is 53.8 and 57.6 for HumanEval and MBPP, respectively, but 507.6 for CodeScope (Generation). So, some LLMs that do well in HumanEval and MBPP, such as WizardCoder, do not do well in CodeScope (Generation).

In CodeScope (Overall), we see that the rankings of LLMs on CodeScope, HumanEval, and MBPP are not consistent. This shows the advantages of CodeScope in terms of its breadth and challenge. CodeScope tests both code generation and code understanding skills, which are more relevant for real-world programming scenarios. CodeScope also uses multilingual, multidimensional, multitask, and execution-based evaluation methods, which make the evaluation more difficult and diverse. CodeScope simulates the actual programming environment better, and provides a more comprehensive and detailed framework for evaluating the coding skills of LLMs.

[6]HumanEval and MBPP results are from the papers of each model and OpenCompass.

6 Conclusion

We present CodeScope, the first comprehensive benchmark for evaluating LLMs on coding tasks. CodeScope covers 43 programming languages, eight coding tasks, and three evaluation dimensions, using a fine-grained, execution-based evaluation method. We evaluate and analyze eight popular LLMs on CodeScope, and reveal their strengths and weaknesses on different tasks and settings. We also compare CodeScope with other benchmarks, and show the importance of CodeScope in testing LLMs on real-world programming scenarios with multitasking, multilingual, and multidimensional challenges. We offer a comprehensive resource, tool, and benchmark for evaluating LLMs on code understanding and generation skills, aiming to advance future research in this area.

Limitations

Data independence and fairness are paramount when evaluating large language models through benchmarks. However, data leakage is a likely problem for benchmarks for evaluating large language models. While data leakage is considered an issue that hinders the evaluation of models' generalization ability, in this paper, we re-examine the legitimacy and validity of this issue from the following three perspectives:

Data memorization and recitation represent a unique form of knowledge capability. Traditional model evaluation tends to pay more attention to the model's generalization ability, which is mainly based on the model's scale and the training data's limitations. However, in the current large model environment, although the model exhibits memorization and recitation when dealing with vast pre-trained data (Carlini et al., 2019; Yan and Li, 2022), this behavior actually reflects a special knowledge capability of LLMs. This is not exactly equivalent to the natural generalization ability, but in some situations, it can proficiently aid humans in addressing real-world challenges. Therefore, the unique ability of data memorization and recitation still has evaluation value.

Constructing a fully zero-leakage evaluation dataset is technically unfeasible. Given the multitude of LLMs trained on various diverse pre-training corpora, creating a test dataset that is genuinely independent and completely untouched by any model is extremely difficult, especially when the pre-trained data of many models remains closed-source. In addition, even if we attempt to filter data based on timelines, the knowledge base of LLMs is constantly evolving[7]. A zero-leakage dataset today might be accessible to some models in the future due to model updates. To mitigate the risk of leakage, we constructed the CodeScope task dataset using five independent data sources, aiming to minimize reliance on any single source and diminish the risk of bias in evaluation results.

Furthermore, the community has two distinct ways of handling data leakage in benchmark tests. On the one hand, most studies tend to ignore the risk of data leakage, such as AGIEval (Zhong et al., 2023), a recent high-profile bilingual standardized test evaluation benchmark, the multilingual, multimodal and multilevel evaluation benchmark M3Exam (Zhang et al., 2023), and the interdisciplinary comprehensive Chinese evaluation benchmark CMMLU (Li et al., 2023a). Conversely, some recent benchmarks recognize the problem of data leakage, and they generally believe that this challenge is difficult to avoid completely. For example, SciBench (Wang et al., 2023a), an evaluation benchmark for complex scientific problems, and C-Eval (Huang et al., 2023), an evaluation benchmark for multilevel and multi-discipline Chinese, strive to gather data that is difficult to extract or convert into text to mitigate this problem.

The ability to generalize downstream tasks beyond data memorization. Typically, the pre-training of LLMs relies on unsupervised methods, and their performance in various downstream tasks covers a wide range of scenarios (Li et al., 2023b; Wang et al., 2023b). Even though LLMs might encounter certain datasets during the pre-training phase, the application of these datasets in downstream tasks often differs from the scenarios during pre-training. Therefore, despite the potential data leakage, we are essentially still evaluating the capabilities of LLMs to migrate and generalize across different tasks, rather than just their data memorization abilities.

While data leakage is an unavoidable challenge, we should have a broader and more open-minded perspective when evaluating LLMs. We also need to re-examine and redefine our evaluation criteria and methods to ensure their appropriateness and accuracy.

[7]https://platform.openai.com/docs/models/

References

Toufique Ahmed and Premkumar T. Devanbu. 2022. Few-shot training llms for project-specific code-summarization. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, pages 177:1–177:5. ACM.

Rohan Anil, Andrew M. Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, Eric Chu, Jonathan H. Clark, Laurent El Shafey, Yanping Huang, Kathy Meier-Hellstern, Gaurav Mishra, Erica Moreira, Mark Omernick, Kevin Robinson, Sebastian Ruder, Yi Tay, Kefan Xiao, Yuanzhong Xu, Yujing Zhang, Gustavo Hernández Ábrego, Junwhan Ahn, Jacob Austin, Paul Barham, Jan A. Botha, James Bradbury, Siddhartha Brahma, Kevin Brooks, Michele Catasta, Yong Cheng, Colin Cherry, Christopher A. Choquette-Choo, Aakanksha Chowdhery, Clément Crepy, Shachi Dave, Mostafa Dehghani, Sunipa Dev, Jacob Devlin, Mark Díaz, Nan Du, Ethan Dyer, Vladimir Feinberg, Fangxiaoyu Feng, Vlad Fienber, Markus Freitag, Xavier Garcia, Sebastian Gehrmann, Lucas Gonzalez, and et al. 2023. Palm 2 technical report. *CoRR*, abs/2305.10403.

Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, and Ramesh Nallapati. 2023. Multilingual evaluation of code generation models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

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.

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. Deepcoder: Learning to write programs. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.

Cédric Bastoul. 2004. Code generation in the polyhedral model is easier than you think. In *13th International Conference on Parallel Architectures and Compilation Techniques (PACT 2004), 29 September - 3 October 2004, Antibes Juan-les-Pins, France*, pages 7–16. IEEE Computer Society.

Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin T. Vechev. 2021. Tfix: Learning to fix coding errors with a text-to-text transformer. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 780–791. PMLR.

Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 101–113. ACM.

Leslie Pérez Cáceres, Federico Pagnozzi, Alberto Franzin, and Thomas Stützle. 2017. Automatic configuration of GCC using irace. In *Artificial Evolution - 13th International Conference, Évolution Artificielle, EA 2017, Paris, France, October 25-27, 2017, Revised Selected Papers*, volume 10764 of *Lecture Notes in Computer Science*, pages 202–216. Springer.

Nicholas Carlini, Chang Liu, Úlfar Erlingsson, Jernej Kos, and Dawn Song. 2019. The secret sharer: Evaluating and testing unintended memorization in neural networks. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 267–284. USENIX Association.

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

Shubham Chandel, Colin B. Clement, Guillermo Serrato, and Neel Sundaresan. 2022. Training and evaluating a jupyter notebook data science assistant. *CoRR*, abs/2201.12901.

Chun Chen, Jacqueline Chame, and Mary Hall. 2008. Chill: A framework for composing high-level loop transformations. Technical report, Citeseer.

Dehao Chen, David Xinliang Li, and Tipp Moseley. 2016. Autofdo: automatic feedback-directed optimization for warehouse-scale applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO 2016, Barcelona, Spain, March 12-18, 2016*, pages 12–23. ACM.

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.

Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E Gonzalez, et al. 2023. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality. *See https://vicuna. lmsys. org (accessed 14 April 2023)*.

Ananta Kumar Das, Shikhar Yadav, and Subhasish Dhal. 2019. Detecting code smells using deep learning. In *TENCON 2019 - 2019 IEEE Region 10 Conference (TENCON), Kochi, India, October 17-20, 2019*, pages 2081–2086. IEEE.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. Robustfill: Neural program learning under noisy I/O. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 990–998. PMLR.

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.

Martin Fowler. 1999. *Refactoring - Improving the Design of Existing Code*. Addison Wesley object technology series. Addison-Wesley.

Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. Deepfix: Fixing common c language errors by deep learning. In *Proceedings of the aaai conference on artificial intelligence*, volume 31.

Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010. On the use of automated text summarization techniques for summarizing source code. In *17th Working Conference on Reverse Engineering, WCRE 2010, 13-16 October 2010, Beverly, MA, USA*, pages 35–44. IEEE Computer Society.

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.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with APPS. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2020. Measuring massive multitask language understanding. *CoRR*, abs/2009.03300.

Junjie Huang, Chenglong Wang, Jipeng Zhang, Cong Yan, Haotian Cui, Jeevana Priya Inala, Colin B. Clement, Nan Duan, and Jianfeng Gao. 2022. Execution-based evaluation for data science code generation models. *CoRR*, abs/2211.09374.

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.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics.

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*, pages 437–440. ACM.

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.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-Tau Yih, Daniel Fried, Sida I. Wang, and Tao Yu. 2023. DS-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 18319–18345. PMLR.

Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020*, pages 184–195. ACM.

Haonan Li, Yixuan Zhang, Fajri Koto, Yifei Yang, Hai Zhao, Yeyun Gong, Nan Duan, and Timothy Baldwin. 2023a. CMMLU: measuring massive multitask language understanding in chinese. *CoRR*, abs/2306.09212.

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. 2023b. Starcoder: may the source be with you! *CoRR*, abs/2305.06161.

Tsz On Li, Wenxi Zong, Yibo Wang, Haoye Tian, Ying Wang, Shing-Chi Cheung, and Jeff Kramer. 2023c. Finding failure-inducing test cases with chatgpt. *CoRR*, abs/2304.11686.

Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022a. Competition-level code generation with alphacode. *CoRR*, abs/2203.07814.

Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. 2022b. Automating code review activities by large-scale pre-training. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, pages 1035–1047. ACM.

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, SPLASH 2017, Vancouver, BC, Canada, October 23 - 27, 2017*, pages 55–56. ACM.

Tao Lin, Xue Fu, Fu Chen, and Luqun Li. 2021. A novel approach for code smells detection based on deep leaning. In *Applied Cryptography in Computer and Communications: First EAI International Conference, AC3 2021, Virtual Event, May 15-16, 2021, Proceedings 1*, pages 171–174. Springer.

Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomás Kociský, Fumin Wang, and Andrew W. Senior. 2016. Latent predictor networks for code generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics.

Fan Long and Martin C. Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 166–178. ACM.

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*.

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.

Lech Madeyski and Tomasz Lewowski. 2023. Detecting code smells using industry-relevant data. *Inf. Softw. Technol.*, 155:107112.

Radu Marinescu. 2005. Measurement and quality in object-oriented design. In *21st IEEE International Conference on Software Maintenance (ICSM 2005), 25-30 September 2005, Budapest, Hungary*, pages 701–704. IEEE Computer Society.

Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. 2014. The impact of code review coverage and code review participation on software quality: a case study of the qt, vtk, and ITK projects. In *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*, pages 192–201. ACM.

Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur. 2010. DECOR: A method for the specification and detection of code and design smells. *IEEE Trans. Software Eng.*, 36(1):20–36.

Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2013a. Lexical statistical machine translation for language migration. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 651–654. ACM.

Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013b. Semfix: program repair via semantic analysis. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 772–781. IEEE Computer Society.

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.

OpenAI. 2023. GPT-4 technical report.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA*, pages 311–318. ACL.

Karl Pettis and Robert C. Hansen. 1990. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20-22, 1990*, pages 16–27. ACM.

Dmitry Plotnikov, Dmitry Melnik, Mamikon Vardanyan, Ruben Buchatskiy, and Roman Zhuykov. 2013. An automatic tool for tuning compiler optimizations. In *Ninth International Conference on Computer Science and Information Technologies Revised Selected Papers*, pages 1–7. IEEE.

Mihail Popov, Chadi Akel, Yohan Chatelain, William Jalby, and Pablo de Oliveira Castro. 2017. Piecewise holistic autotuning of parallel programs with CERE. *Concurr. Comput. Pract. Exp.*, 29(15).

Julian Aron Prenner and Romain Robbes. 2021. Automatic program repair with openai's codex: Evaluating quixbugs. *CoRR*, abs/2111.03922.

Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir R. Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, and Ulrich Finkler. 2021. Project codenet: A large-scale AI for code dataset for learning a diversity of coding tasks. *CoRR*, abs/2105.12655.

Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 254–265. ACM.

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.

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.

Mazeiar Salehie, Shimin Li, and Ladan Tahvildari. 2006. A metric-based heuristic framework to detect object-oriented design flaws. In *14th International Conference on Program Comprehension (ICPC 2006), 14-16 June 2006, Athens, Greece*, pages 159–168. IEEE Computer Society.

Tushar Sharma, Pratibha Mishra, and Rohit Tiwari. 2016. Designite: a software design quality assessment tool. In *Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities, BRIDGE@ICSE 2016, Austin, Texas, USA, May 17, 2016*, pages 1–4. ACM.

Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, and Hongbin Sun. 2022. On the evaluation of neural code summarization. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1597–1608. ACM.

Mohammed Latif Siddiq, Joanna C. S. Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinicius Carvalho Lopes. 2023. Exploring the effectiveness of large language models in generating unit tests. *CoRR*, abs/2305.00418.

Jelena Slivka, Nikola Luburic, Simona Prokic, Katarina-Glorija Grujic, Aleksandar Kovacevic, Goran Sladic, and Dragan Vidakovic. 2023. Towards a systematic approach to manual annotation of code smells. *Sci. Comput. Program.*, 230:102999.

Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori L. Pollock, and K. Vijay-Shanker. 2010. Towards automatically generating summary comments for java methods. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, pages 43–52. ACM.

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.

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Trans. Softw. Eng. Methodol.*, 28(4):19:1–19:29.

Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. 2022. Using pre-trained models to boost code review automation. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 2291–2302. ACM.

Xiaoxuan Wang, Ziniu Hu, Pan Lu, Yanqiao Zhu, Jieyu Zhang, Satyen Subramaniam, Arjun R. Loomba, Shichang Zhang, Yizhou Sun, and Wei Wang. 2023a. Scibench: Evaluating college-level scientific problem-solving abilities of large language models. *CoRR*, abs/2307.10635.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023b. Codet5+: Open code large language models for code understanding and generation. *CoRR*, abs/2305.07922.

Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 364–374. IEEE.

David Williams-King and Junfeng Yang. 2019. Codemason: Binary-level profile-guided optimization. In *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*, pages 47–53.

Zhuokui Xie, Yinghao Chen, Chen Zhi, Shuiguang Deng, and Jianwei Yin. 2023. Chatunitest: a chatgpt-based automated unit test generation tool. *CoRR*, abs/2305.04764.

Weixiang Yan and Yuanchun Li. 2022. Whygen: Explaining ml-powered code generation by referring to training examples. In *44th IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2022, Pittsburgh, PA, USA, May 22-24, 2022*, pages 237–241. ACM/IEEE.

Weixiang Yan, Yuchen Tian, Yunzhe Li, Qian Chen, and Wen Wang. 2023. Codetransocean: A comprehensive multilingual benchmark for code translation. *CoRR*, abs/2310.04951.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, pages 440–450. Association for Computational Linguistics.

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.

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.

Wenxuan Zhang, Sharifah Mahani Aljunied, Chang Gao, Yew Ken Chia, and Lidong Bing. 2023. M3exam: A multilingual, multimodal, multilevel benchmark for examining large language models. *CoRR*, abs/2306.05179.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023a. Judging llm-as-a-judge with mt-bench and chatbot arena.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023b. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *CoRR*, abs/2303.17568.

Wanjun Zhong, Ruixiang Cui, Yiduo Guo, Yaobo Liang, Shuai Lu, Yanlin Wang, Amin Saied, Weizhu Chen, and Nan Duan. 2023. Agieval: A human-centric benchmark for evaluating foundation models.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2023. Language agent tree search unifies reasoning acting and planning in language models. *CoRR*, abs/2310.04406.

Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K. Reddy. 2022. Xlcost: A benchmark dataset for cross-lingual code intelligence.

A Appendix

A.1 Related Work

Other Benchmarks Puri et al. (2021) propose the semantics-based CodeNet benchmark, which significantly increases the variety of supported programming languages, yet the evaluation tasks remain relatively limited as code similarity and classification, and code translation. It is noteworthy that experts find about half of the solutions in the CodeNet datasets are incorrect (Zhu et al., 2022). Hao et al. (2022) introduce AiXBench, which includes 175 Java samples. However, since each sample lacks unit tests, model performance has to be evaluated manually. MultiPL-E (Cassano et al., 2022) translates the HumanEval and MBPP benchmarks into eighteen languages through compiler methods. However, the translation accuracy is not guaranteed. ClassEval (Du et al., 2023) evaluates LLMs in the complex scenario of class-level program synthesis, including 100 class-level Python program synthesis samples. The study indicates that current LLMs still face considerable challenges in effectively handling class-level code generation.

Code Summarization Code summarization research typically transitions from initial template-based methods to more advanced Neural Machine Translation models. Template-based methods, while relying on expert knowledge, often fall short in accurately capturing code semantics (Sridhara et al., 2010; Haiduc et al., 2010). In contrast, NMT-based models, exemplified by CodeNN (Iyer et al., 2016), incorporate techniques such as AST flattening and Graph Neural Networks to achieve a deeper understanding of source code (Shi et al., 2022; LeClair et al., 2020).

Code Smell Detecting and repairing code smells early in the development process is essential to enhance the reliability, scalability, and maintainability of software systems. Fowler (1999) first proposes the concept of code smells, introducing 22 types of code smells that violate design rules, along with their features and impacts. Traditional code smell detection mainly adopts metric-based and rule/heuristic-based approaches. Metric-based approaches combine metrics such as complexity, coupling, and class size, and then use thresholds or ranges to determine the presence of code smells (Marinescu, 2005; Salehie et al., 2006). Rule/Heuristic-based approaches rely on rules and heuristic criteria set by experienced developers or experts (Moha et al., 2010; Sharma et al., 2016). In recent years, researchers have been using deep learning to explore methods for detecting code smells. Lin et al. (2021) use a fully convolutional network that focuses on code semantic features for detection, while the convolutional neural network trained by Das et al. (2019) demonstrated commendable efficacy in detecting specific code smells.

Code Review McIntosh et al. (2014) demonstrate that code review effectively reduces the defect rate of software. With the rapid development of deep learning, recent studies focus on using deep learning to automate the process of code review. Tufano et al. (2022) propose a method based on the T5 model that can automatically provide code improvement suggestions for reviewers, while automatically making code changes based on submitted code and natural language review feedback. This approach shows great potential in shortening code review cycles and assisting code submitters. Li et al. (2022b) design four pre-trained tasks specifically for code review, thereby improving the accuracy of code review. At the same time, the performance of neural networks in code review is evaluated based on three tasks: code change quality estimation, code review generation, and code refinement.

Automated Testing In recent studies, Siddiq et al. (2023) explore the ability of LLMs to generate unit tests for software, and evaluate the quality of unit tests generated by LLMs. Li et al. (2023c) introduce differential prompting, using ChatGPT to identify test cases that can trigger program errors. Yuan et al. (2023) propose ChatTESTER to enhance ChatGPT's ability to generate high-quality test cases, and investigate the correctness and usability of test cases generated by ChatGPT, effectively improving the accuracy and efficiency of automated testing. Xie et al. (2023) design a ChatGPT-based automated unit test Generation-Validation-Repair framework called ChatUniTest, which can not only generate high-coverage unit tests, but also repair syntactic and compilation errors.

Program Synthesis Previous works (Balog et al., 2017; Ling et al., 2016; Yin and Neubig, 2017) usually focus on synthesizing and analyzing programs in domain-specific language. Deepcoder (Balog et al., 2017) leverages an encoder-decoder network to predict program properties with given input and outputs. Ling et al. (2016); Yin and Neubig (2017) use RNNs and Ptr-Nets to map natural language descriptions to code elements (e.g., code structure, syntax tree). Devlin et al. (2017) achieve directly generating target codes by applying a seq-to-seq generative network.

Code Translation Code translation involves taking source code written in one programming language (the source language) and generating equivalent code in another (the target language). Most of the existing works only focus on mutual translation between two languages. One of the most popular benchmarks, CodeXGLUE (Lu et al., 2021) provides CodeTrans involving the translation between Java and C#. While Ahmed and Devanbu (2022); Nguyen et al. (2013a) include Java & Python, and Java & C#, respectively. For enabling translation among various programming languages, the works by Zhu et al. (2022); Yan et al. (2023); Khan et al. (2023) primarily focus on supporting 7, 45, and 11 programming languages. For evaluation, most of works (Ahmed and Devanbu, 2022; Lu et al., 2021; Zhu et al., 2022) still rely on n-gram matching like BLEU (Papineni et al., 2002) and CodeBLEU (Ren et al., 2020), which suffer from the heavy dependence on comprehensiveness and accuracy of reference code. Yan et al. (2023); Khan et al. (2023) adopt executable metrics *Debugging Successful Rate@K* (DSR@k) and *Pass@K* that tests the code by executability and accuracy under test cases.

Code Repair The earliest tools for code repair are static analysis tools that check code for basic errors like syntax violations. For automatic code repairing, the semantic-based repairing techniques develop with the help of a specification for the intended program behavior (Nguyen et al., 2013b; Weimer et al., 2009; Long and Rinard, 2015; Qi et al., 2014). With the inspiration of neural machine translation (NMT), some works leverage the LMs to enhance automatic code repair. Tufano et al. (2019) leverage the capabilities of NMT to transform flawed code into corrected code, essentially simulating the fusion of an Abstract Syntax Tree (AST). Prenner and Robbes (2021) explore the CodeX's (Chen et al., 2021) code repairing performance of Python and Java on QuixBugs (Lin et al., 2017). Recently, Khan et al. (2023) test the GPT-3.5's performance over 11 program languages. Unlike the existing works, we evaluate 8 kinds of powerful LLMs code repairing ability cover 14 programming languages, providing a more comprehensive evaluation. TFix (Berabi et al., 2021), a semantic-based dataset for JavaScript code repair. Just et al. (2014) and Gupta et al. (2017) propose execution-based code repair datasets Defects4J and DeepFix for Java and C, respectively.

Code Optimization A wide range of optimization methods are developed regularly, most of which enhance the efficiency of developers' code at the compiler or source code level. Many optimization techniques apply in compilers during the compiling process, including dead code elimination, inline expansion, loop optimization, instruction scheduling, automatic parallelization. Researchers work on using different static techniques to obtain the best compiler flag combinations to gain performance (Cáceres et al., 2017; Popov et al., 2017; Plotnikov et al., 2013). Profile-guided optimization (PGO) approaches (Pettis and Hansen, 1990; Williams-King and Yang, 2019) obtain a profile (feedback data) by executing the code and producing an optimized version of the code by analyzing the profile. However, such a method needs more compiling time and thus needs better usability. Another group of optimization techniques transforms the source code to make it more efficient. Research in this category focuses on optimizing loops using the polyhedral model (Bondhugula et al., 2008; Bastoul, 2004). Some researchers use auto-tuning (Chen et al., 2008, 2016) to generate multiple code variants using alternative algorithms or code transformation (such as loop unrolling, and blocking scheduling) and search for the best optimization.

A.2 The CodeScope Benchmark

A.2.1 Code Summarization

According to the TIOBE Programming Community Index, we collect code summarization data for the 43 most popular programming languages from the Rosetta Code programming website[8]. In order to keep the difficulty of test samples consistent across different languages to control the fairness of the evaluation, we select 170 high-quality programming tasks and extract 4,838 code samples, which prioritize tasks covering a wider range of programming languages, aiming to ensure an equivalent level of task difficulty across all languages. To preserve balance in our dataset, we ensure that the number of samples for each programming language is at least 30. We revise and craft a reference summarization for each sample based on the task description, sample code explanation, and source code. As a result, each sample includes the task description, programming language, source code, and its reference summarization.

A.2.2 Code Smell

Madeyski and Lewowski (2023) provide a large dataset of code smells identified by experienced developers from industry-relevant open source Java projects. Slivka et al. (2023) propose a systematic approach for manually annotating code smells and collect a dataset of C# code smells from active projects on GitHub. In CodeScope, we integrate the above Java and C# datasets, comprising three

[8]The Rosetta Code programming website aims to demonstrate the differences in usage between languages by providing multilingual code solutions to a given set of tasks.

class-level and two method-level code smell categories. We select 100 representative samples for each language. We manually review each sample to guarantee the dataset's balance and high quality, ensuring an equal number of samples for each code smell type. Each sample includes source code, smelly code snippets, and potential code smell options.

A.2.3 Code Review

We use the code quality estimation dataset released by Li et al. (2022b), which includes real-world code changes, quality estimation, and review comment data in Github, covering nine commonly used programming languages, including Python, Java, Go, C++, Javascript, C, C#, PHP, and Ruby. In order to ensure the balance and high quality of the dataset, we filter each language according to the code length and select 200 representative samples for each language.

A.2.4 Automated Testing

We construct an automated testing dataset using samples of four programming languages Python, Java, C, and C++ in the dataset crawled from Codeforces. Each sample consists of a problem description, input and output specifications, input and output samples with explanations, the source code solution, and multiple test cases. To ensure the high quality of our dataset, we manually verify and select 100 representative samples from each language, each exhibiting a 100% pass rate, line coverage, and branch coverage.

Given that the limited token count of LLMs can critically constrain the generation of effective test cases, we limit the number of test cases generated by LLMs to 5 to ensure fairness of evaluation. To establish a benchmark for manual written test cases, we randomly select 5 test cases from each sample and test their pass rate, line coverage, and branch coverage on the source code solution. To reduce the bias caused by random selection, we repeat this process 5 times and take the average of the results.

A.2.5 Program Synthesis

We collect problem descriptions and correct submissions of corresponding problems in 14 different programming languages, including C++, Java, Python, C, C#, Ruby, Delphi, Go, JavaScript, Kotlin, PHP, D, Perl, and Rust.

To ensure the quality of the dataset, we exclude problems with fewer than 10 test cases, as well as non-deterministic problems with multiple potential outputs for the same test input. When selecting ground truth, we perform execution validation and exclude submissions that failed to compile in different environments due to environmental differences, as well as submissions for brute force solutions exceeding 5k tokens.

A.2.6 Code Translation

We utilize the Codeforces4LLM dataset constructed in the program synthesis task. Given that evaluating all permutation combinations across 14 programming languages incurs excessive overhead, we limit the number of code pairs to 15 at each difficulty level. Furthermore, we maintain the integrity of the remaining data within the Codeforces4LLM dataset.

A.2.7 Code Repair

We expand the Codeforces4LLM dataset by collecting additional incorrect code submissions for each problem and execute them in the MultiCodeEngine to obtain code error information. Furthermore, we maintain the integrity of the remaining data within the Codeforces4LLM dataset.

A.2.8 Code Optimization

To ensure that each task had diverse solutions from an algorithmic and source code syntactic perspective, we evaluate the performance of different solutions across various test cases. Therefore, we select problem samples with more than 10 correct answer submissions and over 20 test cases.

In addition, we inspect the execution time and memory usage of code submissions for each problem in its corresponding test cases. Based on the inspection results, we identify the code submission samples with the longest execution time and highest memory usage for each problem. These samples are deemed to possess considerable optimization potential in terms of time and memory efficiency, and we calibrate the time and memory efficiency baseline for each problem. In summary, each data sample includes the problem description, the type of programming language, the code solution flagged for optimization potential concerning execution time and memory usage, and an array of test cases pertinent to the problem.

A.3 Experimental Setup

Closed-sourced LLMs GPT-4 (OpenAI, 2023) and GPT-3.5 are LLMs released by OpenAI that not only generate semantically coherent and logically rigorous natural language text, but also exhibit excellent performance on code understanding and generation tasks. Furthermore, PaLM 2 (Anil et al., 2023), with its 340 billion parameters, has been trained on 3.6 trillion tokens. It encompasses training in 20 programming languages, significantly enhancing its capabilities in generating code.

Open-sourced LLMs LLaMA 2 (Touvron et al., 2023) is a highly regarded open-source regression LLM, trained on 2 trillion tokens and expanded its context length to 4096 tokens. Additionally, the popular Vicuna (Chiang et al., 2023) fine-tunes LLaMA 2 using a dialogue corpus, aiming to process dialogue text with greater precision.

Open-sourced Code LLMs StarCoder (Li et al., 2023b), a widely-adopted open-source Code LLM, is trained on a corpus of 1 trillion tokens from over 80 programming languages and boasts a context length of 8,192 tokens. WizardCoder (Luo et al., 2023), a fine-grained instruction evolution and incorporation of code debugging features as well as space-time complexity constraints, leverage a new training dataset constructed from Code Alpaca to fine-tune StarCoder. The recently released Code LLaMA (Rozière et al., 2023) is trained further on a specific code dataset based on LLaMA 2, capable of stably generating up to 100K context tokens.

To facilitate the replication of our experimental results, we have detailed the specific configuration information for each LLM and the corresponding inference environments in Table 6.

A.4 Case Study

We provide comprehensive case studies for each experiment in Table 35 to Table 50, detailing specific workflows and associated information pertinent to each.

Model	Model Version	Model Size	Inference GPU
GPT-4	gpt-4-0613	-	-
GPT-3.5	gpt-3.5-turbo-0613	-	-
PaLM 2	text-bison-001	-	-
LLaMA 2	LLaMA-2-70b-chat-hf	70B	NVIDIA Tesla A800 * 4
StarCoder	starchat-beta	15B	NVIDIA Tesla A800 * 1
Code LLaMA	Code LLaMA-34b-Instruct-hf	34B	NVIDIA GeForce RTX 4090 * 4
WizardCoder	WizardCoder-15B-V1.0	15B	NVIDIA GeForce RTX 4090 * 2
Vicuna	vicuna-13b-v1.5-16k	13B	NVIDIA GeForce RTX 4090 * 2

Table 6: Configuration information for the baseline LLMs, parameters for the experiment, and hardware information for inference.

Model	BLEU	METEOR	ROUGE	BERTScore	Overall
GPT-4	4.73	19.94	24.23	85.72	**33.66**
GPT-3.5	4.29	19.88	22.72	85.69	33.14
Vicuna	3.39	18.19	22.26	84.40	32.06
WizardCoder	3.29	19.04	21.60	84.01	31.99
Code LLaMA	3.11	17.85	21.80	83.34	31.52
LLaMA 2	2.84	17.41	21.69	83.67	31.40
StarCoder	2.74	17.06	20.72	84.19	31.18
PaLM 2	4.71	19.10	16.17	81.08	30.27

Table 7: Performance comparison of LLMs in code summarization.

Model	Java				C#				Overall
	Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	
WizardCoder	24.00	20.64	24.00	21.87	65.00	82.55	65.00	72.44	**46.94**
LLaMA 2	25.00	33.75	25.00	24.57	41.00	64.86	41.00	50.17	38.17
GPT-3.5	32.00	24.85	32.00	25.11	30.00	86.25	30.00	34.18	36.80
Vicuna	15.00	19.27	15.00	14.07	47.00	75.27	47.00	57.04	36.21
Code LLaMA	21.00	33.37	21.00	25.38	35.00	78.30	35.00	38.22	35.91
PaLM 2	30.00	38.31	30.00	26.29	41.00	32.03	41.00	35.96	34.32
GPT-4	27.00	25.29	27.00	22.00	34.00	57.16	34.00	41.83	33.53
StarCoder	1.00	6.25	1.00	1.72	49.00	74.74	49.00	34.61	27.16

Table 8: Performance comparison of LLMs in code smell.

Model	Accuracy	Precision	Recall	F1-score	BLEU	ROUGE	BERTScore	Overall
Code LLaMA	52.67	55.39	52.67	45.82	0.95	8.40	82.50	**42.63**
Vicuna	52.22	53.60	52.22	47.40	0.77	6.97	82.56	42.25
LLaMA 2	51.22	52.02	51.22	46.06	0.73	7.44	83.17	41.69
GPT-4	49.56	49.51	49.56	48.34	1.02	8.39	83.63	41.43
PaLM 2	47.56	48.59	47.56	47.97	1.62	9.01	83.54	40.84
GPT-3.5	49.89	49.52	49.89	37.93	1.37	8.37	84.52	40.21
StarCoder	47.22	45.99	47.22	42.50	0.46	7.78	83.70	39.27
WizardCoder	49.56	49.42	49.56	36.68	0.36	6.64	81.77	39.14

Table 9: Performance comparison of LLMs in code review.

Model	Python			Java			C			C++			Overall
	PR	LC	BC	PR	LC	BC	PR	LC	BC	PR	LC	BC	
Human	100.0	96.59	93.84	100.0	97.83	92.14	100.0	97.30	93.05	100.0	98.17	94.04	96.91
GPT-3.5	68.20	97.74	95.86	72.80	98.68	93.75	63.60	97.05	92.52	68.00	97.73	93.33	**86.61**
PaLM 2	64.80	94.79	92.92	63.60	97.49	92.55	59.60	96.05	91.25	60.80	97.54	93.76	83.76
LLaMA 2	61.60	96.92	95.85	59.40	96.30	90.67	54.80	96.15	90.90	57.60	98.10	94.15	82.70
Code LLaMA	62.60	95.59	95.00	57.60	92.97	86.70	55.60	94.69	89.04	60.20	98.39	93.70	81.84
WizardCoder	57.00	96.13	94.21	56.40	97.93	91.67	53.20	96.33	90.34	56.60	98.31	93.81	81.83
GPT-4	59.60	96.76	94.03	52.60	95.87	88.71	47.20	95.36	89.33	39.40	97.83	92.24	79.08
StarCoder	52.20	90.51	88.85	54.40	92.87	87.21	47.20	94.68	87.50	55.60	97.77	93.94	78.56
Vicuna	50.60	90.78	87.78	47.60	90.08	81.94	39.80	95.07	88.55	47.60	97.36	89.98	75.59

Table 10: Performance comparison of LLMs in automated testing, where PR denotes Pass Rate, LC denotes Line Coverage, BC denotes Branch Coverage.

Language	Metric	GPT-4	GPT-3.5	Code LLaMA	LLaMA 2	PaLM 2	WizardCoder	Vicuna	StarCoder
	Accuracy	49.00	49.00	52.00	57.00	44.00	50.00	49.00	50.00
	Precision	48.90	41.41	54.15	63.73	45.31	50.53	48.49	25.00
	Recall	49.00	49.00	52.00	57.00	44.00	50.00	49.00	50.00
C	F1-score	47.83	34.54	44.85	51.00	44.30	37.04	44.32	33.33
	BLEU	1.81	2.76	1.72	1.41	2.11	0.48	1.23	0.05
	ROUGE	9.72	10.24	10.65	9.32	10.26	8.67	8.31	1.01
	BERTScore	84.34	85.06	81.04	83.94	84.82	83.01	83.08	75.79
	Accuracy	55.00	51.00	51.00	49.00	46.00	52.00	51.00	49.00
	Precision	55.25	52.21	52.55	48.73	45.83	75.51	51.94	24.75
	Recall	55.00	51.00	51.00	49.00	46.00	52.00	51.00	49.00
C#	F1-score	54.45	43.23	42.21	46.15	45.45	37.63	46.03	32.89
	BLEU	0.40	0.49	0.00	0.38	0.00	0.18	0.77	0.00
	ROUGE	8.32	7.85	8.67	7.64	7.53	6.37	7.10	0.93
	BERTScore	83.74	84.46	83.48	83.41	83.89	82.13	82.64	75.83
	Accuracy	57.00	53.00	48.00	51.00	42.00	51.00	46.00	50.00
	Precision	57.23	75.77	45.27	53.84	42.43	56.99	40.53	25.00
	Recall	57.00	53.00	48.00	51.00	42.00	51.00	46.00	50.00
C++	F1-score	56.65	39.67	39.22	39.88	42.21	45.27	36.89	33.33
	BLEU	1.48	1.62	1.16	0.88	3.10	0.37	0.94	0.08
	ROUGE	7.62	7.20	7.65	7.29	10.22	5.56	6.09	0.81
	BERTScore	83.61	84.28	81.64	83.32	84.03	81.52	82.68	75.50
	Accuracy	54.0	49.00	51.00	46.00	57.00	48.00	67.00	50.00
	Precision	54.60	24.75	53.05	43.75	57.00	24.74	69.22	25.00
	Recall	54.0	49.00	51.00	46.00	57.00	48.00	67.00	50.00
Go	F1-score	52.46	32.89	41.10	40.66	57.00	32.65	66.02	33.33
	BLEU	0.76	1.19	0.70	0.45	1.49	0.16	0.47	0.00
	ROUGE	8.61	8.06	7.75	7.05	9.50	6.01	7.13	0.67
	BERTScore	83.65	84.53	82.00	83.12	84.30	80.22	82.55	75.37
	Accuracy	46.00	47.00	54.00	52.00	46.00	49.00	52.00	50.00
	Precision	45.83	44.12	61.11	53.25	45.89	44.74	52.38	25.00
	Recall	46.00	47.00	54.00	52.00	46.00	49.00	52.00	50.00
Java	F1-score	52.46	32.89	41.10	40.66	57.00	32.65	66.02	33.33
	BLEU	0.76	1.19	0.70	0.45	1.49	0.16	0.47	0.00
	ROUGE	8.61	8.06	7.75	7.05	9.50	6.01	7.13	0.80
	BERTScore	83.65	84.53	82.00	83.12	84.30	80.22	82.55	75.44
	Accuracy	45.00	53.00	51.00	49.00	53.00	50.00	50.00	51.00
	Precision	44.16	75.77	51.33	48.59	55.27	25.00	50.00	75.25
	Recall	45.00	53.00	51.00	49.00	53.00	50.00	50.00	51.00
JavaScript	F1-score	42.94	39.67	47.73	44.99	53.96	33.33	44.30	35.52
	BLEU	1.17	1.16	1.00	0.62	1.55	0.41	0.52	0.00
	ROUGE	9.35	10.39	9.56	7.48	11.20	7.23	7.65	0.99
	BERTScore	83.93	84.77	83.92	83.19	84.35	82.84	83.05	75.89
	Accuracy	44.00	48.00	51.00	48.00	43.00	49.00	52.00	50.00
	Precision	42.56	45.27	52.55	47.40	47.27	24.75	53.70	25.00
	Recall	44.00	48.00	51.00	48.00	43.00	49.00	52.00	50.00
PHP	F1-score	41.15	39.22	42.21	44.82	45.02	32.89	46.77	33.33
	BLEU	0.63	1.37	0.00	0.00	0.57	0.38	0.60	0.04
	ROUGE	7.11	7.35	7.10	5.78	6.16	6.84	6.42	0.76
	BERTScore	83.39	84.46	81.84	82.82	79.84	82.63	82.56	75.43
	Accuracy	48.00	49.00	50.00	51.00	38.00	46.00	55.00	49.00
	Precision	47.77	24.75	50.00	51.96	38.70	36.41	58.86	24.75
	Recall	48.00	49.00	50.00	51.00	38.00	46.00	55.00	49.00
Python	F1-score	46.63	32.89	41.56	44.16	38.28	34.43	49.50	32.89
	BLEU	0.96	1.54	1.18	0.74	2.69	0.65	1.00	0.00
	ROUGE	9.37	9.54	9.29	8.29	10.76	6.93	7.39	0.80
	BERTScore	83.56	84.52	83.54	82.80	84.17	81.49	82.57	75.51
	Accuracy	48.00	50.00	66.00	58.00	59.00	51.00	48.00	49.00
	Precision	47.7	50.00	70.79	64.88	59.46	55.26	43.21	24.75
	Recall	48.00	50.00	66.00	58.00	59.00	51.00	48.00	49.00
Ruby	F1-score	46.26	36.58	63.92	52.51	58.50	38.56	36.86	32.89
	BLEU	0.91	0.97	0.50	1.03	1.78	0.24	0.81	0.05
	ROUGE	7.17	6.79	6.40	7.16	7.18	4.87	5.23	0.73
	BERTScore	83.09	83.91	81.33	82.69	82.91	79.72	81.90	75.59
Overall		41.37	39.97	**42.41**	41.77	40.79	38.29	41.92	34.28

Table 11: Detailed experimental results of code review.

Language\Model	**GPT-4**	**GPT-3.5**	**PaLM 2**	**Code LLaMA**	**WizardCoder**	**LLaMA 2**	**StarCoder**	**Vicuna**	**Avg.**
C++	86.67	66.67	20.00	13.33	10.00	0.00	0.00	6.67	28.10
Java	90.00	60.00	10.00	23.33	13.33	3.33	0.00	0.00	28.57
Python	53.33	33.33	3.33	3.33	0.00	3.33	0.00	0.00	13.81
C	66.67	33.33	3.33	0.00	6.67	3.33	0.00	0.00	16.19
C#	56.67	33.33	3.33	3.33	0.00	0.00	0.00	0.00	13.81
Ruby	36.67	26.67	3.33	3.33	6.67	0.00	3.33	0.00	11.43
Go	56.67	26.67	6.67	6.67	0.00	0.00	0.00	0.00	13.81
JavaScript	23.33	10.00	3.33	0.00	0.00	6.67	0.00	0.00	6.19
Kotlin	76.67	56.67	6.67	6.67	0.00	0.00	3.33	0.00	21.43
PHP	30.00	16.67	0.00	3.33	0.00	0.00	0.00	0.00	7.14
Rust	73.33	53.33	10.00	10.00	3.33	0.00	0.00	0.00	21.43
Perl	70.00	53.33	23.33	10.00	10.00	3.33	6.67	3.33	25.24
D	33.33	33.33	0.00	0.00	0.00	0.00	0.00	0.00	9.52
Delphi	66.67	46.67	6.67	0.00	3.33	0.00	0.00	0.00	17.62
Avg.	**58.57**	39.29	7.14	5.95	3.81	1.43	0.95	0.71	-

Table 12: Evaluation result of program synthesis on ***easy*** problems, employing the PASS@5 metric.

Language\Model	**GPT-4**	**GPT-3.5**	**PaLM 2**	**Code LLaMA**	**WizardCoder**	**LLaMA 2**	**StarCoder**	**Vicuna**	**Avg.**
C++	10.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.43
Java	20.00	10.00	0.00	0.00	0.00	0.00	0.00	0.00	4.29
Python	10.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.43
C	6.67	3.33	0.00	0.00	0.00	0.00	0.00	0.00	1.43
C#	10.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.43
Ruby	10.00	3.33	0.00	0.00	0.00	0.00	0.00	0.00	1.90
Go	16.67	3.33	3.33	0.00	0.00	0.00	0.00	0.00	3.33
JavaScript	0.00	9.09	0.00	0.00	0.00	0.00	0.00	0.00	1.30
Kotlin	17.24	3.45	0.00	0.00	0.00	0.00	0.00	0.00	2.96
PHP	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Rust	13.33	6.67	0.00	0.00	0.00	0.00	0.00	0.00	2.86
Perl	36.67	20.00	6.67	3.33	3.33	0.00	0.00	0.00	10.00
D	3.33	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.48
Delphi	0.00	10.00	0.00	0.00	0.00	0.00	0.00	0.00	1.43
Avg.	**10.99**	4.94	0.71	0.24	0.24	0.00	0.00	0.00	-

Table 13: Evaluation result of program synthesis on ***hard*** problems, employing the PASS@5 metric.

from\to	**C++**	**Java**	**Python**	**C**	**C#**	**Ruby**	**Go**	**JavaScript**	**Kotlin**	**PHP**	**Rust**	**Perl**	**D**	**Delphi**	**Avg.**
C++	-	60.00	26.67	73.33	40.00	60.00	46.67	13.33	33.33	26.67	53.33	0.00	0.00	6.67	33.85
Java	53.33	-	40.00	46.67	33.33	13.33	26.67	13.33	46.67	13.33	40.00	0.00	0.00	0.00	25.13
Python	40.00	20.00	-	33.33	33.33	53.33	46.67	13.33	20.00	33.33	40.00	6.67	0.00	0.00	26.15
C	73.33	66.67	40.00	-	26.67	20.00	33.33	6.67	46.67	13.33	13.33	0.00	0.00	0.00	26.15
C#	53.33	73.33	66.67	20.00	-	53.33	20.00	33.33	40.00	26.67	40.00	0.00	0.00	0.00	32.82
Ruby	26.67	33.33	53.33	6.67	46.67	-	20.00	0.00	26.67	33.33	6.67	0.00	0.00	0.00	19.49
Go	53.33	66.67	66.67	33.33	46.67	20.00	-	6.67	60.00	0.00	33.33	0.00	6.67	0.00	30.26
JavaScript	33.33	26.67	46.67	53.33	40.00	20.00	26.67	-	33.33	13.33	20.00	33.33	0.00	0.00	26.67
Kotlin	40.00	26.67	46.67	13.33	40.00	13.33	6.67	26.67	-	6.67	46.67	6.67	0.00	0.00	21.03
PHP	46.67	60.00	40.00	26.67	53.33	46.67	26.67	26.67	53.33	-	46.67	6.67	0.00	0.00	33.33
Rust	40.00	40.00	46.67	20.00	20.00	26.67	0.00	6.67	26.67	6.67	-	26.67	0.00	0.00	20.00
Perl	33.33	33.33	26.67	20.00	33.33	46.67	26.67	20.00	46.67	26.67	46.67	-	0.00	0.00	27.69
D	33.33	60.00	60.00	66.67	33.33	33.33	26.67	26.67	40.00	53.33	46.67	46.67	-	0.00	40.51
Delphi	66.67	60.00	40.00	46.67	40.00	20.00	26.67	6.67	33.33	26.67	40.00	33.33	26.67	-	35.90
Avg.	45.64	48.21	46.15	35.38	37.44	32.82	25.64	15.38	38.97	21.54	36.41	12.31	2.56	0.51	**28.50**

Table 14: Evaluation result of code translation on ***easy*** problems using GPT-3.5, employing the Pass@1 metric.

from\to	**C++**	**Java**	**Python**	**C**	**C#**	**Ruby**	**Go**	**JavaScript**	**Kotlin**	**PHP**	**Rust**	**Perl**	**D**	**Delphi**	**Avg.**
C++	-	26.67	6.67	20.00	13.33	6.67	6.67	6.67	20.00	0.00	6.67	0.00	0.00	0.00	8.72
Java	26.67	-	20.00	33.33	13.33	0.00	0.00	0.00	13.33	0.00	26.67	0.00	0.00	0.00	10.26
Python	20.00	20.00	-	6.67	46.67	20.00	13.33	13.33	33.33	13.33	13.33	13.33	6.67	0.00	16.92
C	80.00	33.33	0.00	-	13.33	6.67	26.67	0.00	6.67	6.67	6.67	0.00	0.00	0.00	13.85
C#	26.67	20.00	13.33	13.33	-	6.67	6.67	20.00	20.00	13.33	0.00	0.00	0.00	0.00	10.77
Ruby	33.33	33.33	33.33	26.67	20.00	-	20.00	0.00	13.33	26.67	13.33	13.33	6.67	0.00	18.46
Go	13.33	20.00	6.67	13.33	20.00	6.67	-	13.33	13.33	0.00	6.67	0.00	0.00	0.00	8.72
JavaScript	27.27	18.18	0.00	18.18	18.18	0.00	0.00	-	9.09	18.18	0.00	18.18	0.00	0.00	9.79
Kotlin	40.00	40.00	13.33	20.00	13.33	20.00	6.67	0.00	-	6.67	6.67	0.00	0.00	0.00	12.82
PHP	53.85	38.46	30.77	23.08	38.46	38.46	0.00	23.08	30.77	-	23.08	7.69	0.00	0.00	23.67
Rust	13.33	33.33	6.67	20.00	13.33	33.33	0.00	6.67	6.67	6.67	-	0.00	0.00	0.00	10.77
Perl	40.00	33.33	53.33	26.67	26.67	13.33	6.67	13.33	26.67	40.00	26.67	-	6.67	6.67	24.62
D	40.00	26.67	26.67	20.00	13.33	13.33	6.67	0.00	20.00	26.67	13.33	6.67	-	0.00	16.41
Delphi	46.67	20.00	0.00	46.67	6.67	6.67	6.67	0.00	0.00	0.00	6.67	0.00	0.00	-	10.77
Avg.	35.47	27.95	16.21	22.15	19.74	13.21	7.69	7.42	16.40	12.17	11.52	4.55	1.54	0.51	**14.04**

Table 15: Evaluation result of code translation on ***hard*** problems using GPT-3.5, employing the Pass@1 metric.

from\to	**C++**	**Java**	**Python**	**C**	**C#**	**Ruby**	**Go**	**JavaScript**	**Kotlin**	**PHP**	**Rust**	**Perl**	**D**	**Delphi**	**Avg.**
C++	-	86.67	46.67	53.33	53.33	53.33	60.00	33.33	66.67	33.33	80.00	0.00	6.67	0.00	44.10
Java	66.67	-	33.33	46.67	40.00	26.67	33.33	40.00	60.00	20.00	46.67	0.00	0.00	0.00	31.79
Python	66.67	60.00	-	33.33	53.33	60.00	53.33	80.00	46.67	53.33	53.33	6.67	0.00	0.00	43.59
C	66.67	80.00	46.67	-	53.33	33.33	60.00	40.00	46.67	40.00	53.33	0.00	0.00	0.00	40.00
C#	73.33	46.67	60.00	53.33	-	60.00	26.67	40.00	53.33	60.00	53.33	0.00	0.00	0.00	40.51
Ruby	40.00	53.33	40.00	20.00	46.67	-	13.33	53.33	53.33	40.00	40.00	6.67	0.00	0.00	31.28
Go	73.33	53.33	46.67	26.67	46.67	26.67	-	53.33	73.33	13.33	40.00	6.67	6.67	6.67	36.41
JavaScript	26.67	53.33	46.67	20.00	33.33	46.67	46.67	-	53.33	20.00	33.33	26.67	13.33	0.00	32.31
Kotlin	73.33	60.00	66.67	6.67	66.67	40.00	20.00	40.00	-	13.33	33.33	20.00	0.00	13.33	34.87
PHP	66.67	53.33	53.33	33.33	60.00	46.67	60.00	60.00	60.00	-	46.67	20.00	6.67	6.67	44.10
Rust	60.00	73.33	26.67	53.33	46.67	26.67	33.33	13.33	40.00	6.67	-	33.33	0.00	20.00	33.33
Perl	66.67	53.33	26.67	40.00	53.33	53.33	33.33	46.67	66.67	53.33	66.67	-	13.33	13.33	45.13
D	66.67	86.67	53.33	33.33	86.67	60.00	40.00	53.33	80.00	60.00	46.67	66.67	-	20.00	57.95
Delphi	60.00	53.33	40.00	40.00	40.00	66.67	46.67	20.00	60.00	26.67	60.00	86.67	26.67	-	48.21
Avg.	62.05	62.56	45.13	35.38	52.31	46.15	40.51	44.10	58.46	33.85	50.26	21.03	5.64	6.15	**40.26**

Table 16: Evaluation result of code translation on ***easy*** problems using GPT-4, employing the Pass@1 metric.

from\to	**C++**	**Java**	**Python**	**C**	**C#**	**Ruby**	**Go**	**JavaScript**	**Kotlin**	**PHP**	**Rust**	**Perl**	**D**	**Delphi**	**Avg.**
C++	-	60.00	20.00	40.00	20.00	6.67	33.33	6.67	13.33	6.67	13.33	0.00	0.00	0.00	16.92
Java	40.00	-	33.33	20.00	20.00	13.33	13.33	6.67	13.33	0.00	20.00	0.00	0.00	0.00	13.85
Python	46.67	46.67	-	20.00	46.67	26.67	46.67	26.67	40.00	40.00	33.33	13.33	6.67	6.67	30.77
C	66.67	46.67	6.67	-	26.67	13.33	20.00	13.33	40.00	13.33	6.67	0.00	0.00	0.00	19.49
C#	46.67	13.33	6.67	13.33	-	6.67	13.33	13.33	33.33	26.67	26.67	0.00	0.00	0.00	15.38
Ruby	26.67	26.67	26.67	13.33	33.33	-	20.00	13.33	33.33	26.67	20.00	26.67	0.00	6.67	21.03
Go	20.00	33.33	6.67	6.67	0.00	0.00	-	20.00	33.33	0.00	6.67	0.00	0.00	0.00	9.74
JavaScript	27.27	9.09	27.27	36.36	18.18	9.09	18.18	-	18.18	27.27	0.00	27.27	9.09	0.00	17.48
Kotlin	46.67	46.67	33.33	20.00	33.33	26.67	20.00	26.67	-	13.33	6.67	0.00	0.00	0.00	21.03
PHP	53.85	23.08	61.54	15.38	38.46	53.85	30.77	30.77	30.77	-	30.77	0.00	0.00	0.00	28.40
Rust	40.00	33.33	53.33	13.33	46.67	26.67	13.33	33.33	33.33	13.33	-	0.00	0.00	0.00	23.59
Perl	53.33	53.33	60.00	40.00	60.00	33.33	46.67	60.00	40.00	33.33	60.00	-	20.00	13.33	44.10
D	33.33	40.00	6.67	26.67	26.67	20.00	40.00	20.00	26.67	13.33	26.67	33.33	-	6.67	24.62
Delphi	60.00	53.33	20.00	33.33	13.33	20.00	20.00	13.33	0.00	6.67	20.00	13.33	13.33	-	22.05
Avg.	43.16	37.35	27.86	22.95	29.49	19.71	25.82	21.85	27.36	16.97	20.83	8.76	3.78	2.56	**22.03**

Table 17: Evaluation result of code translation on ***hard*** problems using GPT-4, employing the Pass@1 metric.

from\to	**C++**	**Java**	**Python**	**C**	**C#**	**Ruby**	**Go**	**JavaScript**	**Kotlin**	**PHP**	**Rust**	**Perl**	**D**	**Delphi**	**Avg.**
C++	-	33.33	13.33	20.00	0.00	0.00	6.67	0.00	6.67	13.33	6.67	0.00	0.00	0.00	7.69
Java	20.00	-	0.00	13.33	0.00	0.00	6.67	0.00	6.67	13.33	0.00	0.00	0.00	0.00	4.62
Python	20.00	13.33	-	0.00	13.33	13.33	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	4.62
C	33.33	13.33	6.67	-	0.00	6.67	0.00	0.00	6.67	13.33	0.00	0.00	0.00	0.00	6.15
C#	20.00	6.67	26.67	6.67	-	13.33	0.00	0.00	0.00	6.67	6.67	0.00	0.00	0.00	6.67
Ruby	0.00	0.00	6.67	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.51
Go	20.00	6.67	13.33	20.00	6.67	6.67	-	0.00	26.67	0.00	0.00	0.00	0.00	0.00	7.69
JavaScript	13.33	13.33	13.33	6.67	0.00	0.00	0.00	-	6.67	0.00	0.00	6.67	0.00	0.00	4.62
Kotlin	0.00	0.00	6.67	0.00	0.00	6.67	0.00	0.00	-	0.00	6.67	0.00	0.00	0.00	1.54
PHP	13.33	6.67	13.33	20.00	0.00	20.00	0.00	0.00	20.00	-	0.00	0.00	0.00	0.00	7.18
Rust	6.67	20.00	6.67	6.67	6.67	6.67	0.00	0.00	6.67	0.00	-	0.00	0.00	0.00	4.62
Perl	6.67	6.67	13.33	0.00	0.00	6.67	0.00	6.67	6.67	6.67	0.00	-	0.00	0.00	4.10
D	20.00	26.67	13.33	13.33	26.67	6.67	6.67	0.00	13.33	6.67	13.33	6.67	-	0.00	11.79
Delphi	26.67	26.67	26.67	13.33	0.00	0.00	6.67	0.00	6.67	6.67	0.00	0.00	0.00	-	8.72
Avg.	15.38	13.33	12.31	9.23	4.10	6.67	2.05	0.51	8.21	5.13	2.56	1.03	0.00	0.00	**5.75**

Table 18: Evaluation result of code translation on ***easy*** problems using StarCoder, employing the Pass@1 metric.

from\to	**C++**	**Java**	**Python**	**C**	**C#**	**Ruby**	**Go**	**JavaScript**	**Kotlin**	**PHP**	**Rust**	**Perl**	**D**	**Delphi**	**Avg.**
C++	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	6.67	0.00	0.00	0.00	0.00	0.00	0.51
Java	6.67	-	0.00	0.00	0.00	0.00	0.00	0.00	13.33	0.00	0.00	0.00	0.00	0.00	1.54
Python	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	6.67	0.00	0.00	0.00	0.00	0.51
C	33.33	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	6.67	0.00	0.00	0.00	0.00	3.08
C#	6.67	6.67	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.03
Ruby	6.67	0.00	20.00	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	6.67	0.00	0.00	2.56
Go	0.00	0.00	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00	6.67	0.00	0.00	0.00	0.51
JavaScript	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-	9.09	0.00	0.00	0.00	0.00	0.00	0.70
Kotlin	13.33	6.67	0.00	6.67	0.00	6.67	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	2.56
PHP	0.00	0.00	7.69	7.69	7.69	0.00	0.00	0.00	7.69	-	0.00	0.00	0.00	0.00	2.37
Rust	0.00	6.67	0.00	0.00	0.00	0.00	0.00	0.00	6.67	0.00	-	0.00	0.00	0.00	1.03
Perl	0.00	13.33	13.33	6.67	6.67	0.00	0.00	6.67	0.00	13.33	0.00	-	0.00	6.67	5.13
D	13.33	6.67	0.00	20.00	0.00	0.00	0.00	0.00	6.67	0.00	0.00	0.00	-	0.00	3.59
Delphi	0.00	6.67	0.00	0.00	0.00	0.00	6.67	0.00	0.00	0.00	0.00	0.00	0.00	-	1.03
Avg.	6.15	3.59	3.16	3.16	1.10	0.51	0.51	0.51	3.86	2.05	0.51	0.51	0.00	0.51	**1.87**

Table 19: Evaluation result of code translation on ***hard*** problems using StarCoder, employing the Pass@1 metric.

from\to	**C++**	**Java**	**Python**	**C**	**C#**	**Ruby**	**Go**	**JavaScript**	**Kotlin**	**PHP**	**Rust**	**Perl**	**D**	**Delphi**	**Avg.**
C++	-	33.33	13.33	26.67	0.00	6.67	6.67	6.67	6.67	0.00	6.67	0.00	0.00	0.00	8.21
Java	26.67	-	26.67	6.67	0.00	6.67	13.33	0.00	6.67	13.33	0.00	0.00	0.00	0.00	7.69
Python	0.00	13.33	-	6.67	0.00	13.33	0.00	0.00	0.00	13.33	0.00	6.67	0.00	0.00	4.10
C	46.67	20.00	13.33	-	0.00	0.00	0.00	0.00	0.00	6.67	0.00	0.00	0.00	0.00	6.67
C#	6.67	20.00	20.00	0.00	-	20.00	0.00	0.00	0.00	6.67	0.00	0.00	0.00	0.00	5.64
Ruby	0.00	0.00	13.33	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.03
Go	20.00	20.00	13.33	20.00	0.00	20.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	7.18
JavaScript	0.00	6.67	13.33	6.67	0.00	6.67	0.00	-	0.00	0.00	6.67	0.00	0.00	0.00	3.08
Kotlin	0.00	6.67	13.33	0.00	6.67	6.67	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	2.56
PHP	20.00	20.00	26.67	0.00	0.00	20.00	0.00	0.00	6.67	-	0.00	0.00	0.00	0.00	7.18
Rust	20.00	6.67	0.00	13.33	0.00	13.33	13.33	0.00	13.33	13.33	-	0.00	0.00	0.00	7.18
Perl	0.00	13.33	0.00	6.67	0.00	0.00	0.00	0.00	13.33	13.33	0.00	-	0.00	0.00	3.59
D	6.67	6.67	13.33	0.00	0.00	13.33	6.67	6.67	6.67	13.33	0.00	6.67	-	0.00	6.15
Delphi	20.00	6.67	0.00	13.33	0.00	0.00	6.67	0.00	0.00	0.00	0.00	0.00	0.00	-	3.59
Avg.	12.82	13.33	12.82	7.69	0.51	9.74	3.59	1.03	4.10	6.15	1.03	1.03	0.00	0.00	**5.27**

Table 20: Evaluation result of code translation on ***easy*** problems using PaLM 2, employing the Pass@1 metric.

from\to	**C++**	**Java**	**Python**	**C**	**C#**	**Ruby**	**Go**	**JavaScript**	**Kotlin**	**PHP**	**Rust**	**Perl**	**D**	**Delphi**	**Avg.**
C++	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Java	6.67	-	0.00	6.67	0.00	0.00	0.00	0.00	13.33	0.00	0.00	0.00	0.00	0.00	2.05
Python	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C	40.00	13.33	0.00	-	0.00	0.00	6.67	0.00	0.00	0.00	0.00	0.00	0.00	0.00	4.62
C#	6.67	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00	6.67	0.00	0.00	0.00	0.00	1.03
Ruby	0.00	6.67	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.51
Go	0.00	13.33	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.03
JavaScript	9.09	0.00	0.00	9.09	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	1.40
Kotlin	6.67	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.51
PHP	0.00	7.69	7.69	7.69	0.00	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00	1.78
Rust	6.67	0.00	0.00	0.00	0.00	0.00	0.00	0.00	6.67	0.00	-	0.00	0.00	0.00	1.03
Perl	0.00	6.67	13.33	0.00	6.67	6.67	0.00	6.67	6.67	20.00	0.00	-	0.00	0.00	5.13
D	13.33	20.00	0.00	13.33	6.67	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-	0.00	4.10
Delphi	6.67	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-	0.51
Avg.	7.37	5.21	1.62	2.83	1.03	0.51	0.51	0.51	2.05	2.05	0.00	0.00	0.00	0.00	**1.69**

Table 21: Evaluation result of code translation on ***hard*** problems using PaLM 2, employing the Pass@1 metric.

from\to	**C++**	**Java**	**Python**	**C**	**C#**	**Ruby**	**Go**	**JavaScript**	**Kotlin**	**PHP**	**Rust**	**Perl**	**D**	**Delphi**	**Avg.**
C++	-	33.33	6.67	40.00	0.00	13.33	20.00	13.33	6.67	6.67	0.00	0.00	0.00	0.00	10.77
Java	20.00	-	0.00	13.33	26.67	6.67	6.67	0.00	6.67	20.00	0.00	0.00	0.00	0.00	7.69
Python	20.00	26.67	-	6.67	13.33	20.00	0.00	0.00	13.33	20.00	0.00	0.00	0.00	0.00	9.23
C	33.33	20.00	13.33	-	6.67	6.67	13.33	0.00	6.67	13.33	0.00	0.00	0.00	0.00	8.72
C#	20.00	33.33	40.00	6.67	-	6.67	0.00	0.00	0.00	40.00	0.00	0.00	0.00	0.00	11.28
Ruby	6.67	6.67	6.67	0.00	0.00	-	0.00	0.00	0.00	13.33	0.00	0.00	0.00	0.00	2.56
Go	26.67	0.00	20.00	26.67	6.67	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	6.15
JavaScript	13.33	26.67	13.33	6.67	6.67	0.00	0.00	-	6.67	6.67	0.00	6.67	0.00	0.00	6.67
Kotlin	0.00	13.33	20.00	6.67	13.33	6.67	0.00	0.00	-	6.67	0.00	0.00	0.00	0.00	5.13
PHP	20.00	20.00	40.00	26.67	6.67	33.33	6.67	0.00	26.67	-	0.00	0.00	0.00	0.00	13.85
Rust	13.33	13.33	26.67	26.67	6.67	13.33	13.33	0.00	6.67	0.00	-	13.33	0.00	0.00	10.26
Perl	6.67	13.33	6.67	6.67	6.67	13.33	0.00	13.33	6.67	6.67	6.67	-	0.00	0.00	6.67
D	33.33	53.33	20.00	20.00	20.00	6.67	6.67	0.00	6.67	20.00	0.00	13.33	-	0.00	15.38
Delphi	20.00	26.67	13.33	6.67	6.67	0.00	6.67	0.00	20.00	13.33	6.67	0.00	0.00	-	9.23
Avg.	17.95	22.05	17.44	14.87	9.23	9.74	5.64	2.05	8.21	12.82	1.03	2.56	0.00	0.00	**8.83**

Table 22: Evaluation result of code translation on ***easy*** problems using WizardCoder, employing the Pass@1 metric.

from\to	**C++**	**Java**	**Python**	**C**	**C#**	**Ruby**	**Go**	**JavaScript**	**Kotlin**	**PHP**	**Rust**	**Perl**	**D**	**Delphi**	**Avg.**
C++	-	6.67	0.00	20.00	0.00	6.67	6.67	0.00	0.00	0.00	6.67	0.00	0.00	0.00	3.59
Java	6.67	-	0.00	6.67	0.00	0.00	0.00	0.00	6.67	0.00	6.67	0.00	0.00	0.00	2.05
Python	0.00	6.67	-	0.00	6.67	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.03
C	46.67	0.00	6.67	-	0.00	6.67	6.67	0.00	0.00	0.00	0.00	0.00	0.00	0.00	5.13
C#	6.67	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00	13.33	0.00	0.00	0.00	0.00	1.54
Ruby	13.33	6.67	13.33	0.00	6.67	-	0.00	0.00	0.00	13.33	0.00	0.00	0.00	0.00	4.10
Go	6.67	6.67	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.03
JavaScript	0.00	9.09	0.00	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.70
Kotlin	6.67	26.67	0.00	6.67	6.67	0.00	0.00	0.00	-	6.67	0.00	0.00	0.00	0.00	4.10
PHP	7.69	15.38	7.69	0.00	7.69	7.69	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00	3.55
Rust	0.00	0.00	0.00	0.00	0.00	6.67	0.00	0.00	6.67	0.00	-	0.00	0.00	0.00	1.03
Perl	0.00	6.67	26.67	6.67	26.67	6.67	13.33	0.00	6.67	33.33	6.67	-	0.00	0.00	10.26
D	13.33	20.00	0.00	6.67	0.00	0.00	0.00	0.00	6.67	6.67	0.00	0.00	-	0.00	4.10
Delphi	13.33	0.00	6.67	6.67	0.00	6.67	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-	2.56
Avg.	9.31	8.04	4.69	4.10	4.18	3.16	2.05	0.00	2.05	5.64	1.54	0.00	0.00	0.00	**3.20**

Table 23: Evaluation result of code translation on ***hard*** problems using WizardCoder, employing the Pass@1 metric.

from\to	C++	Java	Python	C	C#	Ruby	Go	JavaScript	Kotlin	PHP	Rust	Perl	D	Delphi	Avg.
C++	-	20.00	6.67	13.33	0.00	0.00	20.00	0.00	0.00	0.00	0.00	0.00	6.67	0.00	5.13
Java	13.33	-	0.00	6.67	0.00	0.00	6.67	0.00	0.00	6.67	0.00	0.00	0.00	0.00	2.56
Python	33.33	6.67	-	0.00	0.00	13.33	13.33	0.00	0.00	20.00	0.00	0.00	0.00	0.00	6.67
C	40.00	13.33	6.67	-	0.00	6.67	6.67	0.00	0.00	6.67	6.67	0.00	0.00	0.00	6.67
C#	6.67	13.33	0.00	0.00	-	0.00	6.67	0.00	0.00	6.67	0.00	0.00	0.00	0.00	2.56
Ruby	6.67	0.00	6.67	0.00	0.00	-	0.00	0.00	0.00	6.67	0.00	0.00	0.00	0.00	1.54
Go	20.00	20.00	6.67	0.00	6.67	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	4.10
JavaScript	6.67	6.67	6.67	0.00	6.67	0.00	0.00	-	0.00	13.33	0.00	6.67	0.00	0.00	3.59
Kotlin	20.00	0.00	13.33	6.67	0.00	0.00	6.67	0.00	-	6.67	0.00	0.00	0.00	0.00	4.10
PHP	6.67	20.00	33.33	0.00	6.67	6.67	13.33	0.00	20.00	-	0.00	0.00	0.00	0.00	8.21
Rust	13.33	13.33	0.00	13.33	6.67	0.00	0.00	0.00	0.00	6.67	-	0.00	0.00	0.00	4.10
Perl	0.00	6.67	20.00	13.33	0.00	0.00	6.67	0.00	0.00	13.33	0.00	-	0.00	0.00	4.62
D	6.67	6.67	20.00	13.33	6.67	20.00	0.00	6.67	20.00	13.33	0.00	6.67	-	0.00	9.23
Delphi	13.33	26.67	6.67	0.00	6.67	6.67	6.67	0.00	0.00	6.67	0.00	0.00	0.00	-	5.64
Avg.	14.36	11.79	9.74	5.13	3.08	4.10	6.67	0.51	3.08	8.21	0.51	1.03	0.51	0.00	**4.91**

Table 24: Evaluation result of code translation on ***easy*** problems using Code LLaMA, employing the Pass@1 metric.

from\to	C++	Java	Python	C	C#	Ruby	Go	JavaScript	Kotlin	PHP	Rust	Perl	D	Delphi	Avg.
C++	-	0.00	6.67	0.00	0.00	0.00	6.67	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.03
Java	6.67	-	0.00	6.67	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.03
Python	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	6.67	0.00	0.00	0.00	0.00	0.51
C	20.00	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.54
C#	0.00	13.33	0.00	6.67	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.54
Ruby	6.67	0.00	0.00	0.00	6.67	-	0.00	0.00	0.00	6.67	0.00	0.00	0.00	0.00	1.54
Go	0.00	0.00	0.00	0.00	0.00	0.00	-	6.67	0.00	0.00	0.00	0.00	0.00	0.00	0.51
JavaScript	9.09	9.09	0.00	0.00	9.09	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	2.10
Kotlin	0.00	6.67	0.00	0.00	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.51
PHP	7.69	0.00	15.38	0.00	7.69	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00	2.37
Rust	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	6.67	-	0.00	0.00	0.00	0.51
Perl	0.00	6.67	20.00	0.00	0.00	6.67	0.00	6.67	13.33	20.00	0.00	-	0.00	6.67	6.15
D	6.67	0.00	0.00	6.67	0.00	6.67	0.00	0.00	6.67	0.00	0.00	0.00	-	0.00	2.05
Delphi	20.00	0.00	0.00	0.00	0.00	0.00	0.00	6.67	0.00	0.00	0.00	0.00	0.00	-	2.05
Avg.	5.91	2.75	3.23	1.54	1.80	1.03	0.51	1.54	1.54	3.08	0.00	0.00	0.00	0.51	**1.67**

Table 25: Evaluation result of code translation on ***hard*** problems using Code LLaMA, employing the Pass@1 metric.

from\to	C++	Java	Python	C	C#	Ruby	Go	JavaScript	Kotlin	PHP	Rust	Perl	D	Delphi	Avg.
C++	-	13.33	0.00	20.00	0.00	6.67	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	3.08
Java	0.00	-	6.67	6.67	0.00	0.00	0.00	0.00	0.00	6.67	0.00	0.00	0.00	0.00	1.54
Python	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C	46.67	0.00	0.00	-	0.00	0.00	13.33	0.00	0.00	0.00	0.00	0.00	0.00	0.00	4.62
C#	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Ruby	0.00	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Go	6.67	0.00	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.51
JavaScript	0.00	0.00	0.00	6.67	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.51
Kotlin	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00
PHP	0.00	0.00	0.00	0.00	0.00	6.67	6.67	0.00	0.00	-	0.00	0.00	0.00	0.00	1.03
Rust	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00
Perl	0.00	0.00	0.00	6.67	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-	0.00	0.00	0.51
D	13.33	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-	0.00	1.03
Delphi	26.67	0.00	0.00	6.67	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-	2.56
Avg.	7.18	1.03	0.51	3.59	0.00	1.03	1.54	0.00	0.00	0.51	0.00	0.00	0.00	0.00	**1.10**

Table 26: Evaluation result of code translation on ***easy*** problems using LLaMA 2, employing the Pass@1 metric.

from\to	**C++**	**Java**	**Python**	**C**	**C#**	**Ruby**	**Go**	**JavaScript**	**Kotlin**	**PHP**	**Rust**	**Perl**	**D**	**Delphi**	**Avg.**
C++	-	6.67	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.51
Java	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Python	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C	13.33	0.00	0.00	-	0.00	0.00	6.67	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.54
C#	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Ruby	0.00	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Go	0.00	0.00	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
JavaScript	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Kotlin	6.67	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.51
PHP	0.00	7.69	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00	0.59
Rust	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00
Perl	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00
D	6.67	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-	0.00	0.51
Delphi	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-	0.00
Avg.	2.05	1.10	0.00	0.00	0.00	0.00	0.51	0.00	0.00	0.00	0.00	0.00	0.00	0.00	**0.26**

Table 27: Evaluation result of code translation on ***hard*** problems using LLaMA 2, employing the Pass@1 metric.

from\to	**C++**	**Java**	**Python**	**C**	**C#**	**Ruby**	**Go**	**JavaScript**	**Kotlin**	**PHP**	**Rust**	**Perl**	**D**	**Delphi**	**Avg.**
C++	-	0.00	6.67	0.00	6.67	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.03
Java	0.00	-	0.00	6.67	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.51
Python	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C	40.00	0.00	6.67	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	3.59
C#	6.67	0.00	6.67	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.03
Ruby	0.00	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Go	0.00	0.00	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
JavaScript	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Kotlin	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00
PHP	6.67	0.00	6.67	6.67	6.67	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00	2.05
Rust	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00
Perl	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00
D	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	6.67	-	0.00	0.51
Delphi	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-	0.00
Avg.	4.10	0.00	2.05	1.03	1.03	0.00	0.00	0.00	0.00	0.00	0.00	0.51	0.00	0.00	**0.62**

Table 28: Evaluation result of code translation on ***easy*** problems using Vicuna, employing the Pass@1 metric.

from\to	**C++**	**Java**	**Python**	**C**	**C#**	**Ruby**	**Go**	**JavaScript**	**Kotlin**	**PHP**	**Rust**	**Perl**	**D**	**Delphi**	**Avg.**
C++	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Java	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Python	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C	26.67	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	2.05
C#	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Ruby	0.00	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Go	0.00	0.00	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
JavaScript	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Kotlin	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00
PHP	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00
Rust	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00
Perl	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00
D	6.67	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-	0.00	0.51
Delphi	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-	0.00
Avg.	2.56	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	**0.18**

Table 29: Evaluation result of code translation on ***hard*** problems using Vicuna, employing the Pass@1 metric.

Language\Model	**GPT-4**	**GPT-3.5**	**PaLM 2**	**WizardCoder**	**Code LLaMA**	**Vicuna**	**StarCoder**	**LLaMA 2**	**Avg.**
C++	63.33	43.33	10.00	13.33	6.67	10.00	10.00	3.33	20.00
Java	66.67	23.33	10.00	0.00	6.67	6.67	3.33	0.00	14.58
Python	40.00	10.00	3.33	0.00	0.00	0.00	0.00	3.33	7.08
C	30.00	23.33	16.67	3.33	10.00	10.00	0.00	0.00	11.67
C#	40.00	16.67	10.00	3.33	0.00	3.33	3.33	0.00	9.58
Ruby	30.00	6.67	6.67	13.33	6.67	6.67	3.33	10.00	10.42
Go	50.00	10.71	17.86	7.14	3.57	3.57	3.57	0.00	12.05
JavaScript	37.93	24.14	3.45	0.00	6.90	3.45	3.45	0.00	9.91
Kotlin	50.00	15.00	5.00	0.00	0.00	0.00	0.00	5.00	9.38
PHP	10.00	10.00	3.33	3.33	3.33	0.00	0.00	0.00	3.75
Rust	62.96	14.81	0.00	3.70	3.70	0.00	3.70	0.00	11.11
Perl	46.67	36.67	3.33	10.00	10.00	3.33	0.00	0.00	13.75
D	40.00	13.33	13.33	6.67	0.00	0.00	0.00	0.00	9.17
Delphi	46.67	10.00	0.00	3.33	0.00	0.00	0.00	0.00	7.50
Avg.	43.87	18.43	7.36	4.82	4.11	3.36	2.19	1.55	-

Table 30: Evaluation result of code repair on ***easy*** problems, employing the DSR@1 metric.

Language\Model	**GPT-4**	**GPT-3.5**	**PaLM 2**	**WizardCoder**	**StarCoder**	**Code LLaMA**	**Vicuna**	**LLaMA 2**	**Avg.**
C++	6.67	6.67	6.67	0.00	6.67	3.33	3.33	0.00	4.17
Java	23.33	6.67	6.67	10.00	6.67	3.33	0.00	3.33	7.50
Python	10.00	3.33	3.33	3.33	0.00	0.00	0.00	0.00	2.50
C	23.33	26.67	16.67	16.67	6.67	16.67	6.67	0.00	14.17
C#	13.33	6.67	3.33	6.67	0.00	3.33	0.00	3.33	4.58
Ruby	16.67	0.00	16.67	0.00	8.33	4.17	4.17	4.17	6.77
Go	7.69	7.69	7.69	7.69	0.00	0.00	7.69	0.00	4.81
JavaScript	14.29	14.29	0.00	14.29	0.00	0.00	0.00	0.00	5.36
Kotlin	11.76	17.65	17.65	11.76	11.76	0.00	5.88	0.00	9.56
PHP	0.00	8.33	8.33	8.33	0.00	8.33	0.00	0.00	4.17
Rust	19.23	7.69	7.69	7.69	3.85	7.69	3.85	7.69	8.17
Perl	20.00	3.33	0.00	0.00	0.00	0.00	0.00	0.00	2.92
D	20.00	0.00	5.00	0.00	5.00	0.00	0.00	0.00	3.75
Delphi	3.33	3.33	0.00	0.00	0.00	0.00	0.00	0.00	0.83
Avg.	13.55	8.02	7.12	6.17	3.50	3.35	2.26	1.32	-

Table 31: Evaluation result of code repair on ***hard*** problems, employing the DSR@1 metric.

Language	Metric	GPT-4	GPT-3.5	Vicuna	WizardCoder	Code LLaMA	LLaMA 2	StarCoder	PaLM 2
Python	BLEU	4.9	5.05	3.9	2.79	3.48	2.98	2.57	3.28
	METEOR	25.14	24.56	23.82	21.5	22.28	23.2	21.54	15.4
	ROUGE	20.21	21.38	19.13	18.04	18.06	18.08	17.01	17.47
	BERTScore	85.79	85.95	85.11	82.3	83.75	84.53	83.87	76.02
C	BLEU	4.72	4.17	3.83	3.32	2.64	2.83	2.65	4.8
	METEOR	25.43	23.58	23.73	22.51	21.77	22.68	21.95	15.38
	ROUGE	20.02	19.78	19.1	19.35	17.23	17.53	16.67	18.33
	BERTScore	85.71	85.65	84.63	83.0	83.47	84.0	84.48	81.78
C++	BLEU	4.87	4.01	3.24	3.48	2.91	2.66	2.47	4.54
	METEOR	25.06	23.74	22.52	23.06	21.49	21.8	22.15	16.32
	ROUGE	20.51	19.9	18.44	18.9	16.81	17.51	17.3	19.87
	BERTScore	85.87	85.8	85.0	84.82	83.25	84.37	84.6	83.8
Java	BLEU	4.77	4.63	3.81	3.08	2.87	3.03	2.93	3.86
	METEOR	25.28	24.84	24.4	23.33	23.19	23.09	23.19	13.24
	ROUGE	20.13	20.39	19.57	19.53	17.99	18.54	18.02	17.08
	BERTScore	85.83	86.1	85.59	83.6	82.88	84.35	84.69	78.45
C#	BLEU	4.76	4.08	3.58	3.3	2.88	2.75	2.08	4.29
	METEOR	24.49	22.62	22.63	21.76	21.79	21.42	20.89	16.33
	ROUGE	20.36	19.46	18.66	19.26	17.85	17.95	16.19	19.99
	BERTScore	85.99	84.84	83.77	81.64	82.42	83.18	82.88	83.41
JavaScript	BLEU	4.87	4.39	3.01	2.79	2.91	2.63	2.45	2.63
	METEOR	24.52	23.91	22.93	21.88	22.56	22.27	21.65	15.51
	ROUGE	19.84	20.3	18.01	18.76	18.11	17.58	17.09	17.31
	BERTScore	85.56	85.71	84.6	84.52	83.6	84.24	84.29	80.0
Visual Basic	BLEU	4.65	3.51	3.65	4.98	3.17	2.63	2.53	3.71
	METEOR	20.19	19.56	20.36	21.39	20.55	20.08	20.26	13.58
	ROUGE	17.61	16.79	16.49	17.95	17.81	16.15	15.72	16.88
	BERTScore	85.2	85.28	84.23	85.75	84.96	84.47	84.56	83.95
Visual Basic (.NET)	BLEU	5.12	4.42	3.88	2.72	3.54	3.04	3.51	5.96
	METEOR	25.59	23.07	24.44	21.79	21.89	22.32	21.32	16.04
	ROUGE	20.48	18.79	18.67	19.98	18.73	18.99	18.61	21.25
	BERTScore	86.43	84.5	83.76	82.87	82.13	83.7	85.27	82.02
SQL	BLEU	3.19	4.76	4.1	1.53	2.5	2.44	1.8	3.91
	METEOR	22.47	21.35	21.74	19.2	21.53	19.61	18.01	17.74
	ROUGE	17.86	19.22	17.0	16.94	17.24	16.44	15.84	19.93
	BERTScore	85.27	85.79	85.28	85.1	84.61	84.09	83.84	82.63
PHP	BLEU	5.12	5.38	4.29	3.74	4.18	3.58	3.13	5.22
	METEOR	24.15	24.12	23.59	22.92	23.09	23.01	21.57	17.18
	ROUGE	21.25	21.46	19.2	20.85	19.75	19.29	17.88	21.4
	BERTScore	86.18	86.15	85.04	85.76	84.65	85.08	84.62	83.63
MATLAB	BLEU	5.73	4.81	3.45	4.39	3.82	3.36	2.97	6.37
	METEOR	23.3	22.19	22.64	21.17	21.39	21.43	21.07	15.84
	ROUGE	20.97	21.53	19.37	19.97	18.25	18.6	19.16	18.29
	BERTScore	86.08	85.95	84.81	83.85	83.06	83.01	85.26	72.98
Fortran	BLEU	5.4	4.92	3.99	3.46	3.79	2.53	3.39	3.64
	METEOR	24.78	23.84	22.82	22.06	22.17	22.12	22.53	12.9
	ROUGE	20.58	20.98	18.66	18.64	18.12	17.06	18.0	16.86
	BERTScore	85.92	86.19	84.04	83.07	81.75	83.38	84.26	74.82
Go	BLEU	4.49	4.53	3.58	3.98	3.27	2.8	3.1	4.38
	METEOR	25.27	23.54	23.72	22.94	22.85	21.5	22.19	14.71
	ROUGE	20.35	20.29	18.82	19.35	18.56	17.63	18.54	18.11
	BERTScore	85.91	85.99	85.13	85.41	84.36	84.7	85.23	77.65
X86 Assembly	BLEU	4.01	3.64	2.86	1.63	3.01	1.58	2.37	5.88
	METEOR	24.03	21.93	22.11	18.69	20.3	20.78	19.6	14.44
	ROUGE	18.24	18.67	18.03	17.55	15.73	13.43	16.94	17.58
	BERTScore	85.05	85.02	84.94	77.12	78.7	82.84	81.52	79.95
ARM Assembly	BLEU	3.35	3.43	2.34	1.29	1.46	1.03	1.8	4.12
	METEOR	21.16	19.72	17.15	14.06	12.99	13.54	12.78	11.01
	ROUGE	15.36	16.13	13.83	11.38	9.59	7.8	9.04	10.05
	BERTScore	84.81	85.56	75.04	64.66	62.69	67.34	79.18	46.69

Table 32: The detailed performance of each LLM in code summarization across all languages.

Language	Metric	GPT-4	GPT-3.5	Vicuna	WizardCoder	Code LLaMA	LLaMA 2	StarCoder	PaLM 2
Delphi	BLEU	4.38	4.63	3.94	4.27	3.34	2.84	3.12	5.45
	METEOR	23.79	23.4	23.07	22.11	22.72	22.01	21.21	16.38
	ROUGE	20.02	20.73	18.62	19.82	18.47	17.91	17.12	20.8
	BERTScore	85.78	86.12	85.46	85.88	85.03	84.89	84.8	84.33
Ruby	BLEU	5.2	4.68	3.33	2.96	2.53	3.61	2.78	4.4
	METEOR	24.86	23.01	22.16	22.58	23.61	23.6	20.88	16.94
	ROUGE	21.5	20.98	19.34	19.64	19.47	19.59	17.17	19.34
	BERTScore	86.18	86.01	85.14	85.22	85.09	84.85	84.5	82.16
Rust	BLEU	4.48	3.67	3.16	3.6	3.07	2.78	2.55	5.08
	METEOR	24.4	22.54	23.0	22.67	22.07	22.1	21.76	17.82
	ROUGE	19.81	19.44	18.5	19.42	18.15	18.55	17.43	20.51
	BERTScore	85.05	84.71	84.13	84.21	83.66	83.54	84.37	83.62
Swift	BLEU	4.88	3.86	3.35	3.66	2.78	3.33	2.85	5.26
	METEOR	24.27	22.3	21.7	22.42	22.13	22.03	21.62	17.99
	ROUGE	20.13	19.47	17.81	19.4	17.96	17.89	17.24	19.85
	BERTScore	85.68	85.46	84.58	85.4	84.58	84.32	84.59	83.4
R	BLEU	5.64	4.57	3.59	2.72	3.68	3.27	2.62	3.95
	METEOR	26.08	24.08	22.65	22.25	22.65	22.59	20.19	18.81
	ROUGE	21.9	21.14	19.34	19.97	19.47	18.58	17.09	20.25
	BERTScore	86.12	86.17	84.79	85.74	85.11	84.59	84.19	84.95
COBOL	BLEU	4.12	4.24	3.74	2.72	2.86	2.16	2.74	3.58
	METEOR	23.83	22.53	21.21	20.15	20.4	19.47	20.79	13.94
	ROUGE	18.99	19.02	16.92	18.14	16.23	13.97	16.84	18.4
	BERTScore	85.58	85.88	83.13	81.45	81.61	80.71	82.48	80.61
Ada	BLEU	4.78	4.24	3.47	3.15	2.95	2.97	2.56	5.01
	METEOR	24.53	23.28	22.59	22.51	21.8	22.37	21.8	16.38
	ROUGE	20.1	20.02	18.87	19.92	17.26	17.11	17.0	20.97
	BERTScore	85.49	85.95	84.49	85.31	83.5	83.77	84.23	85.02
Julia	BLEU	5.16	4.51	2.97	4.07	3.51	3.22	3.11	4.24
	METEOR	25.21	23.35	21.89	22.34	22.28	22.04	20.6	18.46
	ROUGE	21.39	20.76	18.2	20.11	19.21	18.5	17.26	20.23
	BERTScore	86.13	85.92	84.74	84.56	84.69	84.23	83.74	82.69
SAS	BLEU	5.6	5.18	4.63	2.13	4.26	3.4	3.51	5.12
	METEOR	23.27	21.33	21.65	21.25	22.86	22.75	19.82	19.16
	ROUGE	20.39	20.7	18.71	20.6	19.58	18.3	17.98	20.97
	BERTScore	85.43	85.98	85.15	86.06	85.37	84.5	84.05	85.74
Kotlin	BLEU	4.56	4.01	3.35	3.49	3.02	2.73	2.81	5.71
	METEOR	24.52	23.16	22.99	23.47	22.02	22.29	21.97	16.96
	ROUGE	20.42	19.8	18.91	19.9	17.65	18.94	17.25	20.48
	BERTScore	86.0	85.33	85.27	85.28	84.4	85.16	84.73	83.97
Perl	BLEU	5.18	4.45	3.8	3.63	3.2	2.96	2.61	5.4
	METEOR	23.93	22.93	22.11	22.08	21.81	21.86	20.6	17.05
	ROUGE	20.38	20.66	18.8	19.55	18.14	17.99	16.78	19.76
	BERTScore	85.91	85.8	85.17	85.09	84.29	84.41	84.09	79.91
Objective-C	BLEU	4.31	4.78	3.61	4.54	3.77	3.48	3.65	4.11
	METEOR	23.11	22.46	21.64	19.4	22.23	23.29	21.94	14.61
	ROUGE	19.13	20.04	17.95	18.95	18.22	18.82	18.92	19.07
	BERTScore	85.5	85.74	84.76	84.11	84.75	84.42	83.93	79.18
Prolog	BLEU	4.25	3.42	3.19	2.08	2.38	2.46	2.54	5.15
	METEOR	24.05	23.0	22.96	21.47	21.63	21.98	20.97	16.74
	ROUGE	19.92	19.28	18.6	18.76	16.65	16.37	17.41	19.26
	BERTScore	84.99	84.93	84.1	84.96	82.23	83.35	84.16	82.48
Lua	BLEU	4.81	4.12	3.31	3.8	3.06	2.56	2.2	4.94
	METEOR	24.56	21.96	22.32	21.86	22.11	22.21	20.16	16.07
	ROUGE	20.19	19.6	18.04	19.88	18.45	17.9	16.82	18.9
	BERTScore	85.99	85.98	85.03	85.67	84.96	84.42	84.7	81.39
Scala	BLEU	4.45	4.36	3.32	3.88	2.91	2.87	2.86	4.13
	METEOR	25.35	23.62	23.3	23.09	21.73	22.06	21.42	18.06
	ROUGE	19.55	19.16	17.93	18.92	17.51	17.49	17.2	18.87
	BERTScore	85.77	85.59	84.84	85.41	84.61	84.45	84.53	82.18

Table 33: The detailed performance of each LLM in code summarization across all languages. (Cont. Table 32)

Language	Metric	GPT-4	GPT-3.5	Vicuna	WizardCoder	Code LLaMA	LLaMA 2	StarCoder	PaLM 2
Dart	BLEU	5.06	3.72	3.48	4.17	3.44	3.58	2.66	5.23
	METEOR	25.21	22.62	22.97	21.58	22.04	23.62	22.63	15.62
	ROUGE	20.19	20.71	18.46	18.22	19.66	20.06	18.63	19.67
	BERTScore	86.17	86.03	85.2	85.15	85.45	85.22	85.31	86.26
D	BLEU	5.23	4.4	3.84	3.45	3.5	2.97	3.18	5.89
	METEOR	24.01	22.34	21.78	22.15	22.58	21.71	21.08	16.4
	ROUGE	20.2	20.13	18.22	18.79	17.45	18.08	16.7	20.99
	BERTScore	85.71	85.53	84.83	85.22	84.05	84.58	84.61	83.74
Haskell	BLEU	5.12	4.05	3.27	2.81	2.91	2.44	2.62	4.35
	METEOR	25.79	23.33	22.33	21.85	22.19	21.46	19.31	16.88
	ROUGE	20.66	19.86	18.96	18.52	17.58	17.21	17.01	17.91
	BERTScore	85.55	85.54	84.48	84.2	83.6	83.81	84.29	79.71
VBScript	BLEU	4.14	4.55	3.25	2.99	3.16	2.62	2.91	5.23
	METEOR	23.89	22.05	20.77	20.42	21.7	21.24	21.48	16.57
	ROUGE	19.2	19.59	16.62	18.57	17.81	16.44	17.06	18.58
	BERTScore	85.65	85.85	84.58	85.38	83.79	84.38	84.54	83.81
Scheme	BLEU	4.52	4.05	2.77	2.26	3.2	2.86	2.83	5.62
	METEOR	24.84	23.16	22.82	21.47	21.67	21.25	21.71	17.34
	ROUGE	20.05	20.09	18.21	19.02	17.52	17.48	17.7	20.25
	BERTScore	85.61	85.74	83.8	82.86	82.09	82.79	84.66	81.45
PowerShell	BLEU	5.04	5.47	4.29	3.93	3.58	3.4	3.05	5.11
	METEOR	24.72	23.94	23.88	21.73	23.22	22.33	21.13	15.97
	ROUGE	20.61	21.91	19.46	19.73	19.6	18.51	17.66	19.59
	BERTScore	86.01	86.24	85.29	85.73	85.12	84.69	84.66	82.72
Logo	BLEU	3.45	3.22	2.68	2.07	2.36	2.33	2.64	3.05
	METEOR	21.94	20.0	20.41	19.51	19.8	19.86	20.06	14.9
	ROUGE	17.84	17.99	16.69	18.31	17.34	15.83	17.08	16.92
	BERTScore	85.1	85.05	84.46	85.11	84.36	83.63	84.48	83.5
ABAP	BLEU	4.77	3.59	3.21	4.87	2.1	2.58	2.75	4.8
	METEOR	24.63	20.33	20.75	20.68	21.2	20.55	19.89	15.64
	ROUGE	19.89	18.52	17.11	19.01	17.04	15.83	16.2	18.37
	BERTScore	85.4	85.22	82.06	83.17	81.95	81.38	84.11	80.8
F#	BLEU	5.84	4.84	3.29	3.92	2.67	2.67	2.96	4.35
	METEOR	25.31	23.79	22.91	22.52	22.36	21.88	20.65	17.28
	ROUGE	21.59	21.08	18.57	19.22	18.22	17.21	17.65	19.95
	BERTScore	86.14	86.04	84.68	85.29	84.88	84.3	84.75	81.97
AWK	BLEU	4.02	4.23	3.23	3.85	3.06	2.54	2.22	4.93
	METEOR	23.51	23.53	23.0	22.01	22.77	21.0	19.5	17.03
	ROUGE	19.81	20.09	18.08	20.17	18.56	17.38	16.16	19.96
	BERTScore	85.87	86.07	84.95	85.83	85.06	84.43	83.97	82.11
Groovy	BLEU	4.57	3.87	1.97	3.0	2.93	3.06	2.64	4.35
	METEOR	24.37	22.76	21.18	21.84	23.01	22.91	20.67	17.26
	ROUGE	20.73	20.56	17.94	19.5	18.54	18.46	17.65	19.9
	BERTScore	85.86	85.76	84.56	84.95	85.04	84.52	84.28	82.51
ColdFusion	BLEU	6.45	4.64	1.59	3.79	4.54	3.94	2.43	7.41
	METEOR	24.02	21.65	21.47	22.59	21.46	22.03	16.27	16.78
	ROUGE	21.64	20.93	17.92	22.01	20.05	19.17	16.77	21.83
	BERTScore	86.24	85.91	84.06	86.25	83.25	84.92	84.07	84.21
Zig	BLEU	3.49	3.32	2.45	3.16	2.47	2.58	2.85	4.27
	METEOR	23.22	21.91	19.27	22.57	21.3	21.44	20.32	16.76
	ROUGE	17.58	17.54	16.28	18.29	15.91	16.57	16.0	19.15
	BERTScore	85.46	85.51	84.3	85.57	84.63	84.44	84.8	84.19
Overall		**33.66**	33.14	32.06	31.99	31.52	31.40	31.18	30.27

Table 34: The detailed performance of each LLM in code summarization across all languages. (Cont. Table 33)

Prompt

Please generate a short summarization for the following codes:

```
Align columns - RC - VBScript
Const nr=16, nc=16
ReDim d(nc),t(nr), wor(nr,nc)
i=i+1: t(i) = "Given$a$text$file$of$many$lines,$where$fields$within$a$line$"
i=i+1: t(i) = "are$delineated$by$a$single$'dollar'$character,$write$a$program"
i=i+1: t(i) = "that$aligns$each$column$of$fields$by$ensuring$that$words$in$each$"
i=i+1: t(i) = "column$are$separated$by$at$least$one$space."
i=i+1: t(i) = "Further,$allow$for$each$word$in$a$column$To$be$either$left$"
i=i+1: t(i) = "justified,$right$justified,$or$center$justified$within$its$column."
For r=1 to nr
    If t(r)="" Then Exit For
    w=xRTrim(t(r),"$")
    m=Split(w,"$")
    For c=1 To UBound(m)+1
        wor(r,c)=m(c-1)
        If Len(wor(r,c))>d(c) Then d(c)=Len(wor(r,c))
    Next 'c
    If c>cols Then cols=c
Next 'r
rows=r-1
tt=Array("Left","Right","Center")
For n=1 To 3
    Wscript.Echo
    Wscript.Echo "*****" & tt(n-1) & "*****"
    For r=1 To rows
        w=""
        For c=1 To cols
            x=wor(r,c): s=Space(d(c))
            Select Case n
                Case 1: w=w &"␣"& Left   (x & s,d(c))
                Case 2: w=w &"␣"& Right  (s & x,d(c))
                Case 3: w=w &"␣"& xCentre(x,d(c),"␣")
            End Select 'n
        Next 'c
        Wscript.Echo Mid(w,2)
    Next 'r
Next 'n

Function xCentre(c, n, Pad)
    Dim j
    If n > Len(c) Then
        j = (n - Len(c)) \  2
        If (n - Len(c)) Mod 2 <> 0 Then j = j + 1
        xCentre = Mid(String(j, Pad) & c & String(j, Pad), 1, n)
    Else
        xCentre = c
    End If
End Function 'xCentre

Function xRTrim(c, Pad)
    Dim i2, l, cc
    cc = "": l = Len(c)
    If l > 0 Then
        i2 = l
        Do While (Mid(c, i2, 1) = Pad And i2 > 1)
            i2 = i2 - 1
        Loop
        If i2 = 1 And Mid(c, i2, 1) = Pad Then i2 = 0
        If i2 > 0 Then cc = Mid(c, 1, i2)
    End If
    xRTrim = cc
End Function 'xRTrim
```

Table 35: A detailed case study on high-quality code summarization by GPT-4. (Lang: VBScript).

Generated Summarization
The provided VBScript code is designed to align columns of text from a file. The text file contains many lines, with fields within a line delineated by a single dollar character. The program aligns each column of fields by ensuring that words in each column are separated by at least one space. It also allows for each word in a column to be either left justified, right justified, or center justified within its column. The code includes two functions, xCentre and xRTrim, which are used to center the text and trim the right side of the text respectively.

Reference Summarization
The code reads a text file with lines separated by a dollar character. It aligns each column of fields by ensuring at least one space between words in each column. It also allows each word in a column to be left, right, or center justified. The minimum space between columns is computed from the text, not hard-coded. Trailing dollar characters or consecutive spaces at the end of lines do not affect the alignment. The output is suitable for viewing in a mono-spaced font on a plain text editor or basic terminal.

Analysis
In this example, GPT-4 generates a high-quality summarization for a relatively lower-popularity programming language, VBScript. The generated summarization accurately captures the code's semantics and successfully summarizes its functionality and behavior, achieving high performance in our evaluation metrics (BLEU=25, ROUGE-L=45, METEOR=37, BERTScore=88).

Table 36: A detailed case study on high-quality code summarization by GPT-4. (Cont. Table 35)

Prompt

Please generate a short summarization for the following codes:

```
def readconf(fn):
    ret = {}
    with file(fn) as fp:
        for line in fp:
            # Assume whitespace is ignorable
            line = line.strip()
            if not line or line.startswith('#'): continue

            boolval = True
            # Assume leading ";" means a false boolean
            if line.startswith(';'):
                # Remove one or more leading semicolons
                line = line.lstrip(';')
                # If more than just one word, not a valid boolean
                if len(line.split()) != 1: continue
                boolval = False

            bits = line.split(None, 1)
            if len(bits) == 1:
                # Assume booleans are just one standalone word
                k = bits[0]
                v = boolval
            else:
                # Assume more than one word is a string value
                k, v = bits
            ret[k.lower()] = v
    return ret


if __name__ == '__main__':
    import sys
    conf = readconf(sys.argv[1])
    for k, v in sorted(conf.items()):
        print k, '=', v
```

Generated Summarization

```
def readconf(fn):
    """
    Read a configuration file and return a dictionary of key-value pairs.

    Comments are indicated by '#' at the beginning of a line.
    Boolean values can be specified by a leading ';'.

    Args:
        fn: The name of the configuration file to read.

    Returns:
        A dictionary of key-value pairs.
    """

    ret = {}
    with file(fn) as fp:
        for line in fp:
            # Assume whitespace is ignorable
            line = line.strip()
            if not line or line.startswith('#'): continue
```

Table 37: A detailed case study on low-quality code summarization by PaLM 2 (Lang: Python).

```
            boolval = True
            # Assume leading ";" means a false boolean
            if line.startswith(';'):
                # Remove one or more leading semicolons
                line = line.lstrip(';')
                # If more than just one word, not a valid boolean
                if len(line.split()) != 1: continue
                boolval = False

            bits = line.split(None, 1)
            if len(bits) == 1:
                # Assume booleans are just one standalone word
                k = bits[0]
                v = boolval
            else:
                # Assume more than one word is a string value
                k, v = bits
            ret[k.lower()] = v
    return ret


if __name__ == '__main__':
    import sys
    conf = readconf(sys.argv[1])
    for k, v in sorted(conf.items()):
        print k, '=', v
```

Reference Summarization

The code reads a standard configuration file, ignores lines starting with a hash or semicolon, and blank lines. It sets variables based on the configuration parameters, preserving case sensitivity for parameter data. It also handles optional equals sign used for separating parameter data from the option name and multiple parameters separated by commas. The variables set are 'fullname', 'favouritefruit', 'needspeeling', 'seedsremoved', and 'otherfamily' which is an array.

Analysis

In this example, PaLM 2 generates a low-quality summarization. It misunderstood the task and inserted natural language comments back into the code. As a result, the answers consist of both code language and natural language, which lowers the scores of semantic-based metrics like BERTScore (73.44). However, the generated comments contain similar phrases with the reference summarization, making n-gram overlap-based metrics such as BLEU (38.51) perform well. We observed many PaLM 2 generated cases like this, contributing to the abnormal results in PaLM 2's code summarization overall scores.

Table 38: A detailed case study on low-quality code summarization by PaLM 2. (Cont. Table 37)

Prompt
As an expert software developer with years of experience, please meticulously inspect the following smell code snippet and categorize it into one of the following categories:

- large class - data class - blob - feature envy - long method

The detailed information are as follows:
1. Programming language: Java
2. Smell code snippet:

```
public static int run() {
    List<Integer> integers = Arrays.asList(1, 2, 3, 4);
    List<Integer> mapped = integers.stream().map(n -> n).collect(Collectors.toList()
        );
    return mapped.size();
}
```

3. Source code containing code smells:

```
package basic;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
public class StreamC2 {
    public static void main(String[] args) {
        System.out.println("This␣is␣Java8");
    }
    public static int run() {
        List<Integer> integers = Arrays.asList(1, 2, 3, 4);
        List<Integer> mapped = integers.stream().map(n -> n).collect(Collectors.
            toList());
        return mapped.size();
    }
}
```

Respond only with one of the specified categories.

Generated Smell
long method

Reference Smell
long method

Table 39: A detailed case study on code smell by WizardCoder (Lang: Java).

Prompt
As an expert code reviewer with years of experience, please meticulously inspect the following code change and categorize its quality into one of the following categories:
- 0: Good quality that no review comments required.
- 1: Poor quality that requires review comments.

The detailed information are as follows:
1. Programming language: Javascript
2. Original version code:

```
import resolver from './helpers/resolver';
import {setResolver} from 'ember-mocha';
setResolver(resolver);
mocha.setup({
    timeout: 15000,
    slow: 500
});
```

3. Code diff chunk:

```
@@ -5,7 +5,7 @@ setResolver(resolver);
 mocha.setup({
-    timeout: 15000,
+    timeout: 25000,
     slow: 500
 });
```

Respond only with the number: 0 or 1.

Generated Review (Quality Estimation)
1

Reference Review (Quality Estimation)
1

Table 40: A detailed case study on code review by Code LLaMA (Lang: JavaScript).

Prompt
As an expert code reviewer with years of experience, please meticulously inspect the following code change and provide a concise review comment. The detailed information are as follows:
1. Programming language: Javascript
2. Original version code:

```
import resolver from './helpers/resolver';
import {setResolver} from 'ember-mocha';
setResolver(resolver);
mocha.setup({
    timeout: 15000,
    slow: 500
});
```

3. Code diff chunk:

```
@@ -5,7 +5,7 @@ setResolver(resolver);
 mocha.setup({
-    timeout: 15000,
+    timeout: 25000,
     slow: 500
 });
```

Respond only with a string that represents review comment.

Generated Review (Comments Generation)
The timeout value in the mocha.setup() function has been increased from 15000 to 25000, which may cause longer wait times for test execution. Please ensure that this change is intentional and necessary.

Reference Review (Comments Generation)
Were you having trouble with timeouts in general acceptance tests or only the editor test? It's possible to set timeouts on a per-test basis by using ‘this.timeout(25000)‘ within the ‘it()‘ function. I'd like to drop the global timeout in the future if possible rather than increase it - in some circumstances a failing test can have a knock-on effect which makes all later tests timeout so Travis can get hung up for a long time if no one notices and cancels the build.

Table 41: A detailed case study on code review by Code LLaMA. (Cont. Table 40)

Prompt

As an expert code test developer with years of experience, please provide multiple test cases for a given problem along and its solution. The detailed information are as follows:
1. Problem description: There is the faculty of Computer Science in Berland. In the social net "TheContact!" for each course of this faculty there is the special group whose name equals the year of university entrance of corresponding course of students at the university. Each of students joins the group of his course and joins all groups for which the year of student's university entrance differs by no more than x from the year of university entrance of this student, where x — some non-negative integer. A value x is not given, but it can be uniquely determined from the available data. Note that students don't join other groups. You are given the list of groups which the student Igor joined. According to this information you need to determine the year of Igor's university entrance.
2. Time limit: 1 second
3. Memory limit: 256 megabytes
4. Input specification: The first line contains the positive odd integer n ($1 \leq n \leq 5$) — the number of groups which Igor joined. The next line contains n distinct integers $a_1, a_2, \ldots, a_n$ ($2010 \leq a_i \leq 2100$) — years of student's university entrance for each group in which Igor is the member. It is guaranteed that the input data is correct and the answer always exists. Groups are given randomly.
5. Output specification: Print the year of Igor's university entrance.
6. Sample inputs: ["3\n2014 2016 2015", "1\n2050"]
7. Sample outputs: ["2015", "2050"]
8. Sample explanations: NoteIn the first test the value x=1. Igor entered the university in 2015. So he joined groups members of which are students who entered the university in 2014, 2015 and 2016.In the second test the value x=0. Igor entered only the group which corresponds to the year of his university entrance.
9. Programming language: Python
10. Solution source code:

```
count = int(input())
sum = 0
years = input().split('␣')
for y in years:
    sum += int(y)
print(int(sum/count))
```

Craft 5 test cases with these criteria:
1. Each test case contains a string for both input and output.
2. The solution source code successfully processes the test case's input with no errors.
3. The solution source code's outcome aligns with the test case's output.
4. All test cases are simple and achieve optimal branch and line coverage.
Respond only with a string in the following JSON format: ["input": input string, "output": output string]

Generated Test Cases

```
[   {"input": "1\n2010\n", "output": "2010"},
    {"input": "3\n2014 2016 2015\n", "output": "2015"},
    {"input": "5\n2020 2022 2021 2019 2018\n", "output": "2020"},
    {"input": "2\n2030 2032\n", "output": "2031"},
    {"input": "4\n2070 2072 2071 2069\n", "output": "2070"}  ]
```

Table 42: A detailed case study on automated testing by GPT-3.5 (Lang: Python).

Prompt:
As a professional code developer with years of experience, please provide the corresponding code solution based on the problem description. Detailed information is given below:
1. Problem description: Volodya and Vlad play the following game. There are k pies at the cells of $n \times m$ board. Each turn Volodya moves one pie to the neighbouring (by side) cell. If the pie lies at the border of the board then Volodya can move it outside the board, get the pie and win. After Volodya's move, Vlad bans some edge at the border of the board of length 1 (between two knots of the board) so that Volodya is not able to move the pie outside the board through this edge anymore. The question is: will Volodya win this game? We suppose both players follow the optimal strategy.
2. Input specification: First line contains 3 integers, separated by space: $1 \leq n, m \leq 100$ — dimensions of the board and $0 \leq k \leq 100$ — the number of pies. Each of the next k lines contains 2 integers, separated by space: $1 \leq x \leq n, 1 \leq y \leq m$ coordinates of the corresponding pie. There could be more than one pie at a cell.
3. Output specification: Output only one word: "YES" — if Volodya wins, "NO" — otherwise.
4. Sample inputs: ["2 2 1\n 1 2", "3 4 0", "100 50 2\n 50 25\n 50 25"]
5. Sample outputs: ["YES", "NO", "NO"]
6. Sample explanations: None
7. Programming language: C#
8. Support programming language version: ["MS C#", "Mono C#", ".NET Core C#"]
Respond should only with a string in the following JSON format: {"version": specific version used in the programming language, "target code": the code you produced in the respective programming language version.}

Generated Code:

```
using System;
using System.Linq;

class Program
{
    static void Main(string[] args)
    {
        int[] nm = Console.ReadLine().Split().Select(int.Parse).ToArray();
        int n = nm[0], m = nm[1], k = nm[2];

        bool canWin = false;
        for (int i = 0; i < k; i++)
        {
            int[] xy = Console.ReadLine().Split().Select(int.Parse).ToArray();
            int x = xy[0], y = xy[1];

            if (x <= 5 || y <= 5 || x > n - 5 || y > m - 5)
            {
                canWin = true;
                break;
            }
        }
        Console.WriteLine(canWin ? "YES" : "NO");
    }
}
```

Analysis
Here is a C# solution for the problem. This solution does not require any complex header files, and it is compatible with all versions of C#, including MS C#, Mono C#, and .NET Core C#. It reads the dimensions of the board and the number of pies from the input, then checks the coordinates of each pie. If any pie is close enough to the border (within 5 cells), Volodya can win, because he can move the pie outside before Vlad can block all possible exits.

Table 43: A detailed case study on program synthesis by GPT-4 (Lang: C#; Difficulty: 1900-hard).

Prompt:

As an expert code developer proficient in multiple programming languages with years of experience, please translate the source code in python to the programming language C++ within our supported version. The detailed information is shown as follows:
1. Target programming language: C++
2. Support programming language version: [‘GNU C++11’, ‘GNU C++14’, ‘MS C++’, ‘GNU C++0x’, ‘GNU C++’, ‘MS C++ 2017’, ‘Clang++17 Diagnostics’, ‘GNU C++17’]
3. Source code:

```
n, t = [int(x) for x in input().split()]
l = []
t *= 2
for i in range(n):
    x, a = [int(x) for x in input().split()]
    l.append((x * 2, a * 2))
l.sort()
ans = 0
for i in range(1, n):
    if l[i][0] - l[i - 1][0] > t + (l[i][1] + l[i - 1][1])//2:
        ans += 2
    elif l[i][0] - l[i - 1][0] == t + (l[i][1] + l[i - 1][1])//2:
        ans += 1
print(ans + 2)
```

Respond should only with a string in the following JSON format: {"version": specific version used in the programming language, "target code": the code you produced in the respective programming language version.}

Generated Code:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    int n, t;
    cin >> n >> t;
    vector<pair<int, int>> l;
    t *= 2;
    for (int i = 0; i < n; i++) {
        int x, a;
        cin >> x >> a;
        l.push_back(make_pair(x * 2, a * 2));
    }
    sort(l.begin(), l.end());
    int ans = 0;
    for (int i = 1; i < n; i++) {
        if (l[i].first - l[i - 1].first > t + (l[i].second + l[i - 1].second) / 2) {
            ans += 2;
        }
        else if (l[i].first - l[i - 1].first == t + (l[i].second + l[i - 1].second)
            / 2) {
            ans += 1;
        }
    }
    cout << ans + 2 << endl;
    return 0;
}
```

Table 44: A detailed case study on code translation by GPT-3.5 from Python to C++ (Difficulty: 1200-easy).

Prompt:
As an expert code developer with years of experience, please debug the source code in Delphi based on the corresponding problem description and show the correct code. The detailed information is shown as follows:
1. Problem description: A little boy Gerald entered a clothes shop and found out something very unpleasant: not all clothes turn out to match. For example, Gerald noticed that he looks rather ridiculous in a smoking suit and a baseball cap. Overall, the shop sells n clothing items, and exactly m pairs of clothing items match. Each item has its price, represented by an integer number of rubles. Gerald wants to buy three clothing items so that they matched each other. Besides, he wants to spend as little money as possible. Find the least possible sum he can spend.
2. Input specification: The first input file line contains integers n and m — the total number of clothing items in the shop and the total number of matching pairs of clothing items (). Next line contains n integers a_i ($1 \leq a_i \leq 10^6$) — the prices of the clothing items in rubles.
Next m lines each contain a pair of space-separated integers u_i and v_i ($1 \leq u_i, v_i \leq n, u_i \neq v_i$). Each such pair of numbers means that the u_i-th and the v_i-th clothing items match each other. It is guaranteed that in each pair u_i and v_i are distinct and all the unordered pairs (u_i, v_i) are different.
3. Output specification: Print the only number — the least possible sum in rubles that Gerald will have to pay in the shop. If the shop has no three clothing items that would match each other, print "-1" (without the quotes).
4. Sample inputs: ["3 3\n1 2 3\n1 2\n2 3\n3 1\n", "3 2\n2 3 4\n2 3\n2 1\n"]
5. Sample outputs: ["6\", "-1\"]
6. Programming language: Delphi
7. Buggy code:

```
program Project1;
{$APPTYPE CONSOLE}
uses
  SysUtils,math;
var
  a:array[0..101,0..101] of boolean;
  b:array[0..101] of longint;
  i,j,n,m,x,y,k,ans:longint;
begin
  read(n,m);
  for i:=1 to n do
    read(b[i]);
  for i:=1 to m do
    begin
      read(x,y);
      a[x][y]:=true;
      a[y][x]:=true;
    end;
  ans:=maxlongint;
  for i:=1 to n do
    for j:=1 to n do
      for k:=1 to n do
        if (a[i][j]) and (a[i][k]) and (a[k][j]) then
           ans:=min(ans,b[i]+b[j]+b[k]);
  writeln(ans);
end.
```

8. Error message: WRONG ANSWER
Respond should only with a string in the following JSON format: {"version": specific version used in the programming language, "target code": the code you produced in the respective programming language version.}

Table 45: A detailed case study on code repair by WizardCoder (Lang: Delphi; Difficulty: 1200-easy).

Generated Code:

```
program Project1;
{$APPTYPE CONSOLE}
uses
  SysUtils;
const
  MAX_INT = 2147483647;
var
  a: array[0..101, 0..101] of boolean;
  b: array[0..101] of longint;
  i, j, n, m, x, y, k, ans: longint;
begin
  read(n, m);
  for i := 1 to n do
    read(b[i]);
  for i := 1 to m do
  begin
    read(x, y);
    a[x][y] := true;
    a[y][x] := true;
  end;
  ans := MAX_INT;
  for i := 1 to n do
    for j := 1 to n do
      for k := 1 to n do
        if (a[i][j]) and (a[i][k]) and (a[k][j]) then
          ans := min(ans, b[i] + b[j] + b[k]);
  writeln(ans);
end.
```

Analysis

The error message is due to a missing semicolon at the end of the program. Also, using a constant instead of a variable to store the maximum integer value is better.

Table 46: A detailed case study on code repair by WizardCoder. (Cont. Table 45)

Prompt
As an expert software developer with years of experience, please meticulously inspect the following unoptimized inefficient code and give an optimized version of the code, making it solve the same exact problem while achieving faster execution time. To pass the testcases, the generated optimized code should strictly follow the same input/output format as the original unoptimized code.The detailed information are as follows:
1. Description of the problem: Bajtek is learning to skate on ice. He's a beginner, so his only mode of transportation is pushing off from a snow drift to the north, east, south or west and sliding until he lands in another snow drift. He has noticed that in this way it's impossible to get from some snow drifts to some other by any sequence of moves. He now wants to heap up some additional snow drifts, so that he can get from any snow drift to any other one. He asked you to find the minimal number of snow drifts that need to be created.We assume that Bajtek can only heap up snow drifts at integer coordinates.
2. Programming language: Python 3
3. Unoptimized code:

```
n=int(input())
M=[[0 for i in range(1001)] for i in range(1001)]
ans=n-1
T=[]
for i in range(n) :
    a,b=map(int,input().split())
    M[a][b]=1
    T.append([a,b])
for i in range(n) :
    r=T[i]
    if M[r[0]][r[1]]!=-1  :
        M[r[0]][r[1]]=-1
        l=[[r[0],r[1]]]
        while len(l)>0 :
            g=l[0]
            del(l[0])
            for j in range(n) :
                if T[j][0]==g[0] and M[T[j][0]][T[j][1]]!=-1  or  T[j][1]==g[1] and
                    M[T[j][0]][T[j][1]]!=-1 :
                     l.append([T[j][0],T[j][1]])
                     M[T[j][0]][T[j][1]]=-1
                     ans=ans-1
print(ans)
```

4. Example testcase input: ["2\n2 1\n1 2\n"]
5. Example testcase output: ["1\n"]
Respond only the optimized code in the following JSON format: {"optimized_code": code string}

Table 47: A detailed case study on code optimization to reduce memory usage by GPT-4 (Lang: Python).

Generated Code

```
from collections import defaultdict

def dfs(v, visited):
    visited.add(v)
    for neighbour in graph[v]:
        if neighbour not in visited:
            dfs(neighbour, visited)

n = int(input())
points = [tuple(map(int, input().split())) for \_ in range(n)]
graph = defaultdict(list)
for i in range(n):
    for j in range(i+1, n):
        if points[i][0] == points[j][0] or points[i][1] == points[j][1]:
            graph[points[i]].append(points[j])
            graph[points[j]].append(points[i])

visited = set()
components = 0
for point in points:
    if point not in visited:
        dfs(point, visited)
        components += 1

print(components - 1)
```

Analysis

The above example demonstrates a piece of human-written code aimed at solving a search problem. GPT-4 was tasked with generating an optimized version of the code that achieves more minor memory usage.

In the unoptimized implementation, a 2-dimensional array, "M" (see lines 1-2 in the unoptimized code snippet), is used to represent the adjacency matrix of the snowfield, with 1 denoting a snowdrift and 0 denoting an absence of drift. In the optimized version, GPT-4 creates a dictionary(see lines 9-11 in the optimized code snippet) to store only the coordinates where there is a drift and uses a built-in library called "defaultdict" that efficiently provides a default value for a nonexistent key. Note that the optimization is achieved by reducing the size of the adjacency matrix from $n \times n$ to only include the present points. This makes graph traversal faster and reduces memory usage. By doing so, the optimized code successfully reduced average memory usage by 25KB during runtime, saving an average of 20% memory compared to the unoptimized code and making the implementation more spatially efficient.

Table 48: A detailed case study on code optimization to reduce memory usage by GPT-4. (Cont. Table 47)

Prompt

As an expert software developer with years of experience, please meticulously inspect the following unoptimized inefficient code and give an optimized version of the code, making it solve the same exact problem while achieving faster execution time. To pass the testcases, the generated optimized code should strictly follow the same input/output format as the original unoptimized code. The detailed information are as follows:

1. Description of the problem: One day little Vasya found mom's pocket book. The book had n names of her friends and unusually enough, each name was exactly m letters long. Let's number the names from 1 to n in the order in which they are written. As mom wasn't home, Vasya decided to play with names: he chose three integers i, j, k ($1 \leq i < j \leq n$, $1 \leq k \leq m$), then he took names number i and j and swapped their prefixes of length k. For example, if we take names "CBDAD" and "AABRD" and swap their prefixes with the length of 3, the result will be names "AABAD" and "CBDRD". You wonder how many different names Vasya can write instead of name number 1, if Vasya is allowed to perform any number of the described actions. As Vasya performs each action, he chooses numbers i, j, k independently from the previous moves and his choice is based entirely on his will. The sought number can be very large, so you should only find it modulo $10^9 + 7$.
2. Programming language: Python 3
3. Unoptimized code:

```
import logging
import copy
import sys

logging.basicConfig(stream=sys.stderr, level=logging.DEBUG)

def solve(names):
    m = len(names[0])

    postfix = {}
    for name in names:
        postfix[name[-1:]] = True

    if m == 1:
        return len(postfix)
    newList = list((map(lambda x: x[:-1], names)))

    return len(postfix) * solve(newList)

def main():
    firstLine = input().split()
    firstLine = list(map(int, firstLine))
    inputLines = []
    for i in range(firstLine[0]):
        line = input()
        inputLines.append(line)

    #solve(firstLine)
    print (solve(inputLines) % 1000000007)

def log(*message):
    logging.debug(message)

if __name__ == "__main__":
    main()
```

4. Example testcase input: ["2 3\n AAB\n BAA\n"]
5. Example testcase output: ["4\n"]

Respond only the optimized code in the following JSON format: {"optimized_code": code string}

Table 49: A detailed case study on code optimization to reduce execution time by GPT-4 (Lang: Python).

Generated Code

```
def solve(names):
    m = len(names[0])

    postfix = {}
    for name in names:
        postfix[name[-1:]] = True

    if m == 1:
        return len(postfix)
    newList = list((map(lambda x: x[:-1], names)))

    return len(postfix) * solve(newList)


def main():
    firstLine = input().split()
    firstLine = list(map(int, firstLine))
    inputLines = []
    for i in range(firstLine[0]):
        line = input()
        inputLines.append(line)

    #solve(firstLine)
    print (solve(inputLines) % 1000000007)


if __name__ == "__main__":
    main()
```

Analysis
The above example demonstrates a piece of human-written code aimed at solving a given problem. GPT-3.5 was tasked with generating an optimized version of the code that achieves faster execution time. In the unoptimized code, the developer left her “logging” implementation in the code, which is only for debugging and has nothing to do with achieving the functionality as specified in the problem description. GPT-3.5 successfully accomplished the optimization by simply **removing the debug-related implementation** that takes unnecessary execution time. Moreover, it spotted that the imported package “copy” was never used in the code and **removed the unused package**. By doing so, the optimized code successfully reduced average execution time by 7.28 ms during runtime, saving an average of 37.8% time compared to the unoptimized code and making the implementation more temporally efficient.

Table 50: A detailed case study on code optimization to reduce execution time by GPT-4. (Cont. Table 49)