CodeMMLU: A Multi-Task Benchmark for Assessing Code Understanding Capabilities of CodeLLMs

Dung Nguyen Manh[*], **Thang Phan Chau**[*], **Nam Le Hai**[*†], **Thong T. Doan**[*], **Nam V. Nguyen**[*], **Quang Pham**[*◊], **Nghi D. Q. Bui**[*]
[*]FPT Software AI Center, Viet Nam
[†]Hanoi University of Science and Technology
[◊]VNU-HCM - University of Science
dungnm31@fpt.com, thangpc13@fpt.com, namlh35@fpt.com, thongdt4@fpt.com, namnv78@fpt.com, quangp2808@gmail.com, bdqnghi@gmail.com

Recent advancements in Code Large Language Models (CodeLLMs) have predominantly focused on open-ended code generation tasks, often neglecting the critical aspect of code understanding and comprehension. To bridge this gap, we present CodeMMLU, a comprehensive multiple-choice question-answer benchmark designed to evaluate the depth of software and code understanding in LLMs. CodeMMLU includes over 10,000 questions sourced from diverse domains, encompassing tasks such as code analysis, defect detection, and software engineering principles across multiple programming languages. Unlike traditional benchmarks, CodeMMLU assesses models' ability to reason about code rather than merely generate it, providing deeper insights into their grasp of complex software concepts and systems. Our extensive evaluation reveals that even state-of-the-art models face significant challenges with CodeMMLU, highlighting deficiencies in comprehension beyond code generation. By underscoring the crucial relationship between code understanding and effective generation, CodeMMLU serves as a vital resource for advancing AI-assisted software development, ultimately aiming to create more reliable and capable coding assistants.

GitHub: https://github.com/FSoft-AI4Code/CodeMMLU

1. Introduction

Recent advancements in Code Large Language Models (CodeLLMs) have demonstrated impressive capabilities across various software engineering (SE) tasks (Allal et al., 2023; Bui et al., 2023; Feng et al., 2020; Guo et al., 2024; Li et al., 2023; Lozhkov et al., 2024b; Luo et al., 2023; Nijkamp et al., 2022; Pinnaparaju et al., 2024; Roziere et al., 2023; To et al., 2023; Wang et al., 2021, 2023b; Xu et al., 2022; Zheng et al., 2024c). However, existing benchmarks often fall short in providing rigorous evaluations due to outdated methodologies and potential data leakage (Matton et al., 2024). Moreover, practical applications of CodeLLMs reveal limitations such as bias and hallucination (Liu et al., 2024a; Rahman & Kundu, 2024) that current benchmarks fail to adequately address.

The predominant focus of coding-related benchmarks has been on open-ended, free-form generation tasks, such as code generation/code completion (Austin et al., 2021; Chen et al., 2021; Ding et al., 2023; Hendrycks et al., 2021; Iyer et al., 2018; Lai et al., 2023; Lu et al., 2021; Zhuo et al., 2024) and other SE tasks like program repair Ouyang et al. (2024); Xia et al. (2023) (Table 1). While appealing, these benchmarks struggle to discern whether CodeLLMs truly understand code or merely reproduce memorized training data (Carlini et al., 2022; Nasr et al., 2023). Additionally, the reliance on test cases and executability for evaluation limits the quantity and diversity of these benchmarks across domains, potentially leading to biased and limited generalizations. Recent efforts to improve evaluation through free-form question answering (Li et al., 2024; Liu & Wan, 2021) have introduced

Corresponding author(s): Nghi D. Q. Bui (bdqnghi@gmail.com)

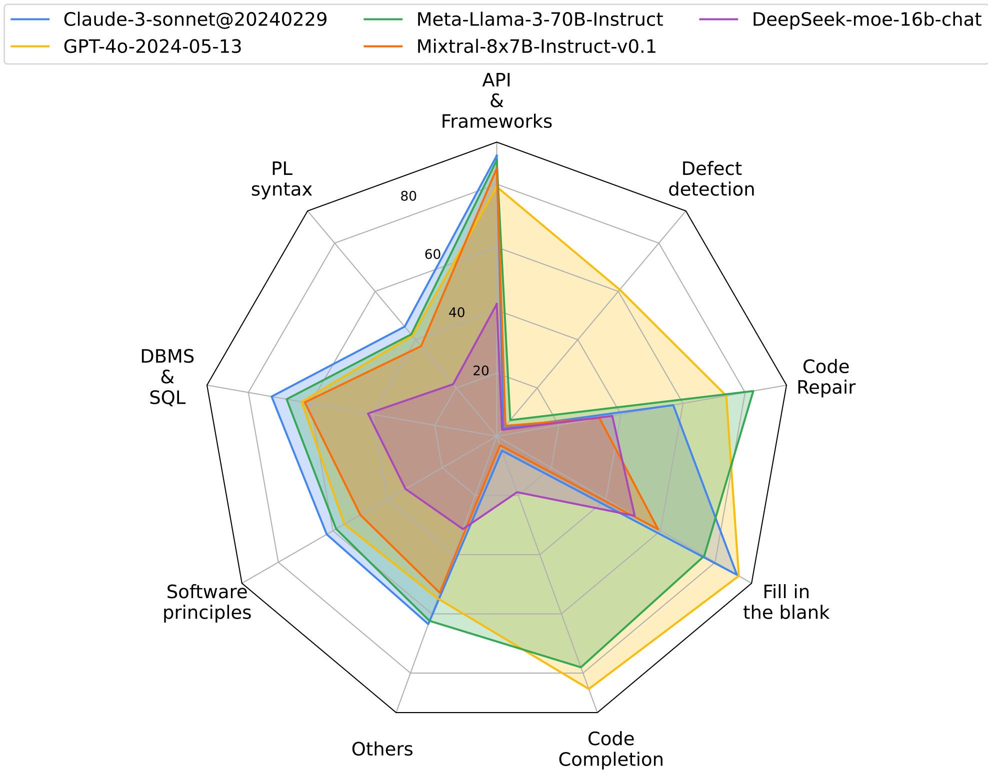

Figure 1 | **Summary performance of LLMs on the CodeMMLU benchmark.** This radar chart presents the evaluation results (accuracy %) of different models across various CodeMMLU tasks.

new challenges, often requiring less rigorous metrics or LLMs-as-a-judge approaches (Zheng et al., 2023). However, LLMs-as-a-judge methods are susceptible to adversarial attacks (Raina et al., 2024), raising concerns about the reliability of such evaluation pipelines for coding tasks.

To address these shortcomings, we introduce CodeMMLU, a novel benchmark designed to evaluate CodeLLMs' ability to understand and comprehend code through multi-choice question answering (MCQA). This approach enables a deeper assessment of how CodeLLMs grasp coding concepts, moving beyond mere generation capabilities. Inspired by the MMLU dataset (Hendrycks et al., 2020) from natural language understanding, CodeMMLU offers a robust and easily evaluable methodology with the following key features:

- **Comprehensiveness:** CodeMMLU comprises over 10,000 questions curated from diverse, high-quality sources, mitigating potential bias from limited evaluation data.
- **Diversity in task, domain, and language:** The dataset covers a wide spectrum of software knowledge, including general QA, code generation, defect detection, and code repair across various domains and more than 10 programming languages.

CodeMMLU enables us to assess LLMs' capabilities in coding and software tasks from a novel perspective, extending beyond traditional code generation and completion. Our analysis reveals several notable findings: (1) previously unexplored bias issues in CodeLLMs, aligning with those observed in natural language MCQA tasks; (2) GPT-4 consistently achieving the highest average performance among closed-source models, while (3) the Meta-Llama family demonstrated the greatest accuracy among open-source models; (4) scaling laws related to model size were partially observed within the same model family but not across different families, suggesting the significant influence of pretraining datasets, methodologies, and model architectures; (5) advanced prompting techniques, such as Chain-of-Thought (CoT), consistently degraded performance, raising concerns about CodeLLMs' reasoning abilities on complex, step-by-step tasks; and (6) benchmarks like HumanEval, when converted from open-ended code generation to MCQA format, show that LLMs perform worse on MCQA, raising concerns about their real capability to understand and comprehend code. These findings highlight the current shortcomings of CodeLLMs and the intricate relationship between model architecture, training data quality, and evaluation methods in determining performance on software-related tasks.

In summary, our key contributions are:

1. We present the first MCQA benchmark for software and coding-related knowledge, addressing the need for diverse evaluation scenarios in the code domain. CodeMMLU enables the evaluation of LLMs' alignment with human inference in the software knowledge domain, similar to advancements in the NLP field.
2. CodeMMLU provides a thorough assessment of LLM capabilities, ensuring a substantial number of samples and diversity across tasks, domains, and languages. This enables a more nuanced understanding of an LLM's strengths and weaknesses, facilitating the development of models better aligned with the complexities and demands of the software domain.
3. Our experiments offer critical insights into LLM performance, highlighting the impact of factors such as model size, model family, and prompting techniques. This provides essential information to the community on effectively utilizing LLMs for specific tasks and domains in software engineering.

2. Related Work

Benchmarks for Code Generation & Understanding The development of Large Language Models (LLMs) for code-related tasks has been accompanied by the creation of diverse benchmark datasets. These benchmarks span a wide range of programming challenges, from basic algorithms to complex software development scenarios. Algorithm-focused benchmarks include HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021), along with their extended versions HumanEval+, MultiPL, and MBPP+ (Liu et al., 2024b). More advanced algorithmic tasks are represented by CodeContests (Li et al., 2022) and LiveCodeBench (Jain et al., 2024), which draw from competitive programming problems. Specialized benchmarks like DS-1000 (Lai et al., 2023) target data manipulation and analysis tasks, while MathQA-Python (Austin et al., 2021) focuses on mathematical problem-solving in Python. Repository-level benchmarks such as RepoBench (Liu et al., 2023), RepoEval (Zhang et al., 2023), and SWE-Bench (Jimenez et al., 2023) simulate real-world software development scenarios. Comprehensive evaluation frameworks like XCodeEval (Khan et al., 2023), CRUXEval (Gu et al., 2024a), and CodeXGLUE (Lu et al., 2021) assess LLMs across multiple dimensions of software development, providing a holistic view of model capabilities.

Programming Comprehension Modeling Research into modeling programmer behavior and cognitive processes has been ongoing since the early days of software development (Shneiderman & Mayer, 1979; Storey, 2005; Xia et al., 2018). Cognitive models aim to describe the mental structures and processes involved in programming, encompassing knowledge, concepts, and techniques used during comprehension and problem-solving. Shneiderman & Mayer (1979) introduced a multi-level model of cognitive structures, distinguishing between semantic and syntactic knowledge. Semantic knowledge includes programming concepts and techniques (e.g., dynamic programming, recursion, sorting methods), while syntactic knowledge relates to programming language grammar (e.g., iteration formats, conditional statements, library functions). To measure programmer comprehension in terms of cognitive processes, Shneiderman & Mayer (1979) proposed five core programming tasks: composition, comprehension, debugging, modification, and learning. This model architecture addresses two crucial questions in programmer comprehension: (1) what knowledge is available to programmers, and (2) what processes do programmers undergo during solution design.

Table 1 | **Comparison between common code understanding benchmarks for LLMs in term of coverage five foundation tasks of programming comprehension model.** † and ⋆ denote the benchmark with the free-flow generation and multiple-choice question answering format, respectively.

Benchmark	Programming Task					#Tasks	Data size
	Programming Knowledge	*Code Composition*	*Code Comprehension*	*Code Modification*	*Code Debugging*		
APPS† Hendrycks et al. (2021)		✓				1	5
MBPP† Austin et al. (2021)		✓				1	974
HumanEval† Chen et al. (2021)		✓				1	164
CRUXEval† Gu et al. (2024b)			✓			2	800
LiveCodeBench† Jain et al. (2024)		✓	✓		✓	4	-
CodeApex⋆† Fu et al. (2023)	✓	✓			✓	3	2.056
CodeMMLU⋆	✓	✓	✓	✓	✓	**6**	**19.912**

Multiple-Choice Question Answering Benchmarks Multiple Choice Questions (MCQs) have emerged as a powerful tool for evaluating the capabilities of Large Language Models (LLMs) across various domains. Recent trends in MCQ-based benchmarks focus on testing advanced reasoning, domain-specific knowledge, and robustness in LLMs, particularly as their capabilities continue to expand (Hendrycks et al., 2020; Lin et al., 2022; Talmor et al., 2019; Zellers et al., 2019). The benefits of MCQs extend beyond enabling large-scale evaluation; they also provide highly reliable results. Compared to open-ended assessment methods (Chiang et al., 2024) that heavily rely on LLM judges or human annotation, MCQs enhance reliability by grounding the knowledge, problem context, and defining possible answers. Despite their convenience, recent studies have revealed that LLMs display significant sensitivity to the order of answer options in MCQs (Robinson et al., 2023; Wang et al., 2023a). This finding underscores the need for appropriate debiasing methods in MCQ benchmarks to address LLM selection bias (Pezeshkpour & Hruschka, 2024; Zheng et al., 2024b). By implementing such methods, researchers can ensure more accurate and fair assessments of LLM performance.

3. CodeMMLU: Data Construction

The CodeMMLU benchmark is constructed to assess large language models' (LLMs) comprehension of programming tasks. We are inspired by programmer comprehension behavior models and integrate multi-leveled cognitive structures and processes to measure the understandability of LLMs on software problems (Shneiderman & Mayer, 1979). CodeMMLU is divided into two primary categories: (i) knowledge-based test sets containing syntactic and semantic tasks, and (ii) real-world programming problems. The overall CodeMMLU structure, as presented in Figure 2, includes distinct approaches for data collection, filtering, and validation in both test sets.

3.1. Knowledge-based task creation

The knowledge-based subset covers a wide range of topics, from high-level software principles concepts to low-level programming language grammars, targets to measuring the LLMs coding capability and comprehensibleness of programming concepts. We collected programming-related MCQs from high-quality platforms, namely GeeksforGeeks, W3Schools, and Sanfoundry (GeeksforGeeks, 2024; SanFoundry, 2024; W3Schools, 2024).

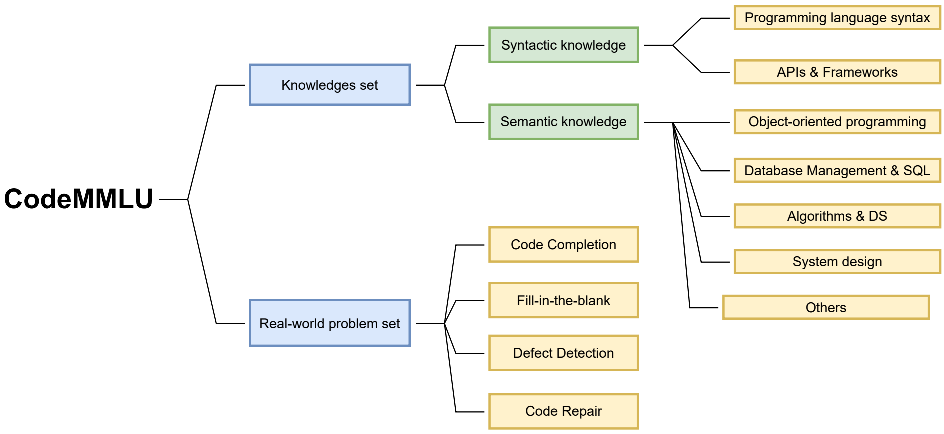

Figure 2 | **Overview of the CodeMMLU Structure.** The CodeMMLU benchmark is divided into two distinct subsets, each designed to evaluate different aspects of LLMs' programming capability.

- **Syntactic set.** Focused on programming language grammar and structural correctness, such as condition statement, format of iteration, common library usage, etc.
- **Semantic set.** Targeted more abstract programming concepts, such as algorithms, data structures, object-oriented principles, etc.

Filtering Process. We manually classified the questions into subjects based on the topics of the collected data. A deep learning-based filtering model was then applied to automatically eliminate low-quality or irrelevant questions. For instance, duplicate or trivial questions that did not sufficiently challenge the code comprehension abilities of LLMs were removed (Appendix A.2.1). The final knowledge-based subset was refined using both manual and deep learning-based filtering to ensure that each question met the desired quality standards, including clarity, lack of ambiguity, and difficulty in evaluating both the semantic and syntactic understanding of LLMs. This filtering process resulted in a knowledge-based subset containing approximately 6,000 syntactic questions and over 3,000 semantic questions, covering 52 topics classified into 5 main subjects (Table 2).

3.2. Task Construction

Our benchmark encompasses five distinct MCQ programming tasks designed to assess the foundational abilities outlined in the cognitive process model of programmer comprehension: Code completion, Code repair, Defect Detection, and Fill in the blank. These tasks cover the core capabilities that any cognitive model of programmer behavior must address: composition, comprehension, debugging, and modification.

Code Completion evaluates a model's composition ability by requiring it to complete partially written code based on provided requirements. We adapted HumanEval (Chen et al., 2021), originally designed for code generation, into an MCQ format. From its 164 unique programming problems, we employed Large Language Models (LLMs) to generate plausible but incorrect solutions as distractors. All options, including correct solutions migrated from HumanEval and generated incorrect ones, were tested for executability. Some incorrect solutions were designed to pass certain test cases but fail others, adding complexity and challenging models to distinguish between correct and nearly-correct solutions based on semantic and syntactic understanding.

Code Repair assesses a model's debugging capability by requiring it to identify and fix errors in provided code snippets. We built this task upon QuixBugs (Lin et al., 2017), which was originally

Table 2 | **Summary of CodeMMLU Subject Categories and Task Distribution.**

	Subject	Topic	Source	Testsize
Syntactic knowledge	API & Frameworks usage	Jquery, Django, Pandas, Numpy, Scipy, Azure, Git, AWS, svg, xml, Bootstrap, NodeJS, AngularJS, React, Vue.	W3Schools, Geeks4Geeks, Sanfoundry	740
	Programming language syntax	C, C#, C++, Java, Javascript, PHP, Python, R, Ruby, MatLab, HTML, CSS, TypeScript.		6,220
Semantic knowledge	DBMS & SQL	DBMS, MySQL, PostgreSQL, SQL.		393
	Software principles	Data structure & Algorithm, Object-oriented programming, Compiler design, Computer organization and Architecture, Software Development & Engineering, System Design.		3,246
	Others	Program accessibility, Computer networks, Computer science, Cybersecurity, Linux, Web technologies, AWS.		1,308
Real-world task	Code completion		HumanEval	163
	Fill in the blank		LeetCode	2,129
	Code repair		QuixBugs	76
	Defect detection		IBM CodeNet	6,006

designed for debugging algorithmic programs. We used a "diff" operation on buggy and corrected versions in QuixBugs (Python and Java) to identify specific fixes, which served as correct solutions. To create plausible distractors, we targeted components frequently involved in bugs (e.g., return statements, loop conditions, if/else/switch expressions) and guided LLMs to generate alternative fixes. These alternatives were designed to seem plausible but not fully resolve the bug. Each distractor was verified for incorrectness, and all options were made executable to ensure that models needed a deep understanding of the code to identify and apply the correct fix.

Defect Detection evaluates a model's ability to identify and understand defects within code snippets, focusing on both logical and syntactical errors. This task measures the comprehension and debugging capabilities of LLMs by requiring them to predict the execution outcome of given code. It includes two sub-tasks: detecting any defects/flaws in the provided code and comprehending the output of a certain test sample. We derived this task set from IBM CodeNet (Puri et al., 2021), a large-scale benchmark for algorithmic coding tasks. We focused on Python and Java subsets, collecting both accepted and buggy versions of code. After filtering out duplicates, we created a diverse set of code samples. For each snippet, we provide the correct execution result (golden answer) and three distracting options, which could be one of several possible outcomes: (i) Compile Error, (ii) Time Limit Exceeded, (iii) Memory Limit Exceeded, (iv) Internal Error, (v) Runtime Error, or (vi) The code does not contain any issue.

Fill in the Blank evaluates a model's code comprehension ability by requiring it to complete missing parts of a code snippet, given documentation and an incomplete code sample. This task assesses not only the model's ability to fill gaps but also its understanding of both high-level programming concepts and low-level grammatical structures. We collected approximately 2,000 coding problems from LeetCode, covering solutions in three widely-used programming languages (Python, Java, C++). From each problem's solution, we randomly selected key components to be blanked out, focusing on elements crucial to the program's logic and flow, such as loop conditions, expression statements, and conditional statements. To create plausible but incorrect options for the multiple-choice question

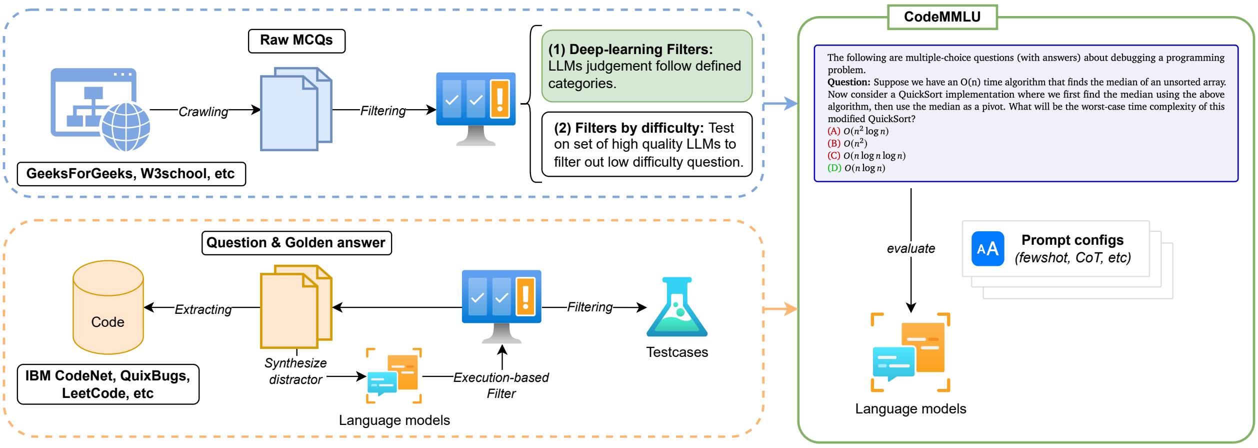

Figure 3 | **Overview of CodeMMLU data creation pipeline.** The blue diagram describe the process of collecting raw multiple-choice questions (MCQs) from open source internet for a knowledge testset. Otherwise, the pipeline of real-world problem indicated in orange area.

(MCQ) format, we employed Large Language Models (LLMs) to generate alternative solutions for the blanked-out components. These distractors were designed to be contextually relevant but incorrect, adding complexity to the task. We executed all generated options to verify their incorrectness, ensuring they do not solve the problem as intended. To further enhance the task's difficulty, we normalized all variable and function names within the code snippets, replacing original names with generic placeholders (e.g., 'var1', 'var2', 'func1'). This normalization process reduces reliance on specific naming conventions, forcing the model to focus purely on the code's logic and structure rather than context clues from variable or function names.

4. Experimental Results

4.1. Setup

Model selection. We evaluate CodeMMLU on 35 on popular open-source CodeLLMs, covering a wide range of parameter sizes and architectures. The models were selected from 7 different families, with parameters ranging from 1 billion to over 70 billion. Each family included base, instructed, and chat versions: MetaLlama3.1/8B/70B (Dubey et al., 2024), MetaLlama3/8B/70B (AI@Meta, 2024), CodeLlaMA/7B/13B/34B (Rozière et al., 2024), DeepSeek-ai/6.7B/7B/33B (Guo et al., 2024), MistralAI/8x7B (Jiang et al., 2024), Qwen2/7B (qwe, 2024), CodeQwen1.5/7B Bai et al. (2023), Yi/6B/9B/34B (AI et al., 2024), StarCoder2/7B/15B (Lozhkov et al., 2024a). In addition to open-source models, we included 3 proprietary models from OpenAI and Claude to ensure comprehensive coverage of the state-of-the-art in language modeling: GPT-3.5/GPT-4 (OpenAI et al., 2024), Claude-3-opus/Claude-3.5-sonnet (The).

Answer extraction. CodeMMLU leverages the MCQ format for scalability and ease of evaluation. In order to maintain this advantage, we only apply simple regex methods to extract the selection answer (i.e. extract by directly answering or contains the pattern "answer is A|B|C|D"). The model response is required to be parsable; otherwise, it will be marked as unanswered.

Prompting. We employed various prompt strategies to test model performance across different scenarios, namely: Zeroshot, Fewshot, CoT, and CoT Fewshot. (Detaill in Appendix A.4.2)

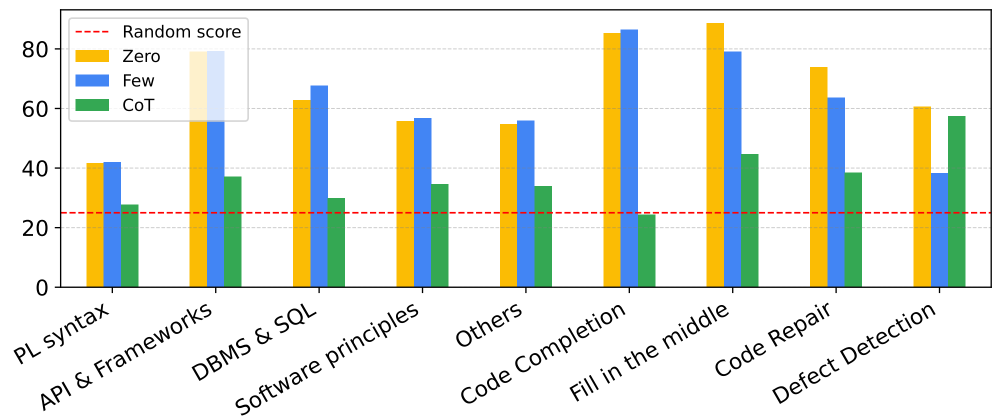

Figure 4 | **Comparison of prompt configuration on GPT-4o.** The experiment exposes the drawback of Chain-of-Thought prompting technique in term of boosting performance on task that not require logic or reasoning.

All experiments were conducted on a cluster of 8 A100 GPUs.

4.2. Key Insights

CodeMMLU revealed significant performance differences across models, as shown in Table 3. OpenAI's GPT-4o outperformed all models on CodeMMLU, demonstrating its quality across diverse tasks (Figure 1). Notably, despite not being the latest model, the instructed version of Meta-Llama-3-70B achieved the highest score among open-source models from 8 families. While LLMs perform well on knowledge-based tasks, they struggle with real-world problems, particularly in defect detection tasks.

Figure 5 illustrates CodeMMLU's capability to measure LLMs' coding knowledge and skills across a wide range of subjects. Our benchmark provides clear, distinct rankings that establish a higher hierarchy of models compared to other code generation benchmarks. Interestingly, the results do not strictly adhere to scaling laws (Kaplan et al., 2020), where larger parameter sizes typically outperform smaller ones. This highlights the impact of data quality in the LLM pretraining process, as recently released models often achieve comparable performance to larger models from previous versions. CodeMMLU also indicates the importance of instruction tuning in improving model performance on complex tasks. Models with instruction tuning substantially outperform their non-instructed counterparts, exemplified by DeepSeek-Coder-33b surpassing its base model by approximately 29%.

Although Chain-of-Thought (CoT) prompting (Wei et al., 2023) is often expected to enhance performance by eliciting deeper reasoning, our experiments reveal its inefficiency across almost all knowledge tasks in the CodeMMLU benchmark. As demonstrated in Figure 4, the significant decline in GPT-4o's performance with CoT suggests that the additional complexity introduced by step-by-step reasoning does not align well with knowledge-seeking tasks (Detail in Appendix A.1). In contrast, few-shot prompting consistently emerges as the most reliable and effective strategy across various tasks, offering a balanced approach without overwhelming the models.

Correlation Between Software Knowledge and Real-World Performance Our experiments revealed a strong correlation between performance on knowledge-based tasks and real-world coding challenges. Specifically, the Pearson correlation score r = 0.61 between model rankings on the knowledge test set and their performance on real-world problems, derived from the accuracy of 43 LLMs across 10 model families, indicates a moderate alignment (Figure 6). This suggests that models

Table 3 | **Summary performance of LLM family on CodeMMLU**. The evaluation results (accuracy %) of different language models across CodeMMLU task.

Family	Model name	Size (B)	Knowledge test Syntactic	Knowledge test Semantic	*Real-world tasks*	CodeMMLU
	Closed-source models					
Anthropic	Claude-3-sonnet@20240229	-	67.22	**66.08**	38.26	53.97
OpenAI	GPT-4o-2024-05-13	-	60.41	57.82	77.18	**67**
	GPT-3.5-turbo-0613	-	61.68	84.88	58.52	51.7
	Open-source models					
Meta Llama	CodeLlama-34b-Instruct-hf	34	56.81	46.93	23.55	38.73
	Meta-Llama-3-70B	70	63.38	86.02	63.1	48.98
	Meta-Llama-3-70B-Instruct	70	64.9	**87.59**	67.68	**62.45**
	Meta-Llama-3.1-70B	70	64.09	87.02	65.65	37.56
	Meta-Llama-3.1-70B-Instruct	70	64.42	87.45	**69.21**	60
Mistral	Mistral-7B-Instruct-v0.3	7	54.42	51.25	31.85	43.33
	Mixtral-8x7B-Instruct-v0.1	46.7	61.17	85.02	61.83	42.96
	Codestral-22B-v0.1	22	60.34	82.17	58.52	47.6
Phi	Phi-3-medium-128k-instruct	14	58.54	54.56	37.89	48.03
	Phi-3-mini-128k-instruct	3.8	53.01	74.18	54.2	37.93
Qwen	Qwen2-57B-A14B-Instruct	57	61.34	57.48	30.48	46.34
	CodeQwen1.5-7B-Chat	7	49.66	67.62	46.06	49.82
Yi	Yi-1.5-34B-Chat	34	58.32	55.59	40.27	49.39
	Yi-1.5-9B-Chat	9	55.64	75.46	60.05	47.23
Deep Seek	DeepSeek-coder-7b-instruct-v1.5	7	56.67	47.9	28.46	41.21
	DeepSeek-coder-33b-instruct	33	53.65	74.89	52.16	36.6
	DeepSeek-moe-16b-chat	16.4	31.74	41.94	41.48	31.01
	DeepSeek-Coder-V2-Lite-Instruct	16	59.91	81.74	64.38	46.51
InternLM	InternLM2-5-20b-chat	20	57.85	55.51	30.44	44.89
StarCoder2	StarCoder2-15b-instruct-v0.1	15	56.58	49.07	42.79	47.94

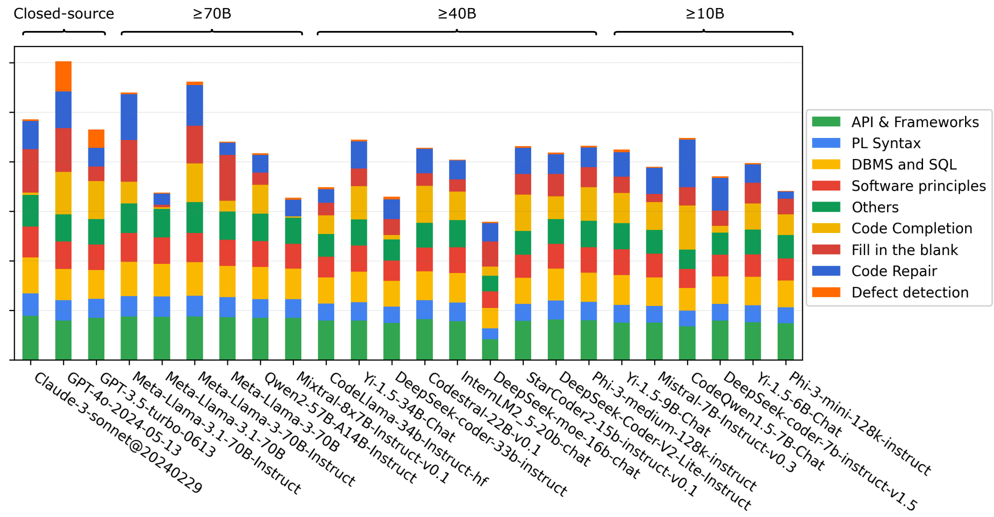

Figure 5 | **CodeMMLU accuracy by task on LLMs.** While knowledge tasks are following the scaling law, real-world tasks offer more challenges to LLMs which indicate the performance of instruction tuning and data quality when evaluating on CodeMMLU.

demonstrating a deeper understanding of software principles consistently excel in real-world coding tasks, highlighting the importance of foundational knowledge for practical coding performance.

Selection bias in MCQs format We experimented evaluating with multiple answer order permutations (follow Zheng et al. (2024a)), the result displayed significant inconsistent behavior exhibited by LLMs when swapping golden answer positions. As presented in Table 8, the model's performance changes dramatically in each answer order configuration, which is based on the correct answer's position. The LLMs accuracy fluctuates between different permutations (i.e. DeepSeek-Coder-34B $\Delta\sigma = 36.66$), demonstrating how sensitive it can be to the structure and order of answers (Figure 7).

Disagreement between Open-ended generation benchmark and MCQ Code completion A notable finding from our experiments is the discrepancy in model performance between open-ended benchmarks and multiple-choice formats. Specifically, when comparing the original HumanEval questions with their multiple-choice equivalents in our CodeMMLU code completion set, we found that models that perform well on HumanEval do not consistently replicate their success in CodeMMLU. For instance, when evaluating identical questions across both formats, the number of cases where models answered both correctly or both incorrectly was unexpectedly low. The correlation scores in Figure 8 further illustrate the weak alignment of success between these two benchmarks, revealing that performance in open-ended tasks does not reliably predict performance in multiple-choice coding tasks.

This lack of alignment suggests that traditional benchmarks might overestimate a model's understanding by focusing too narrowly on code generation. The MCQ format in CodeMMLU forces models to engage with more complex reasoning and contextual understanding, exposing weaknesses that remain hidden in generative tasks.

5. Conclusions

In this work, we introduced CodeMMLU, a comprehensive and scalable benchmark designed to evaluate large language models' (LLMs) capabilities across a wide range of software knowledge

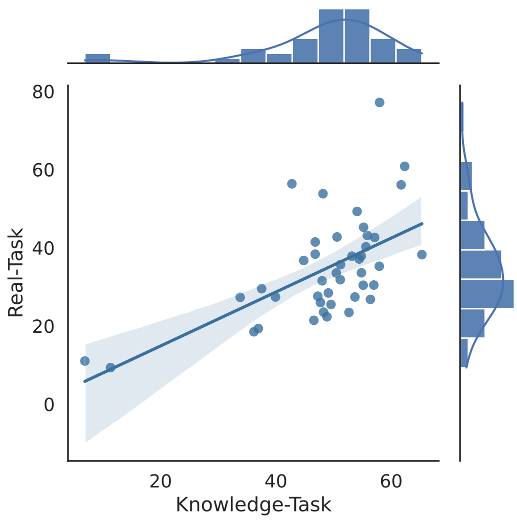

Figure 6 | **Correlation between knowledge test set and real-world problems.** Experiments on 10 LLM families show a clear alignment between models with a strong understanding of software knowledge and their performance on diverse problem-solving tasks in the CodeMMLU real-world task set.

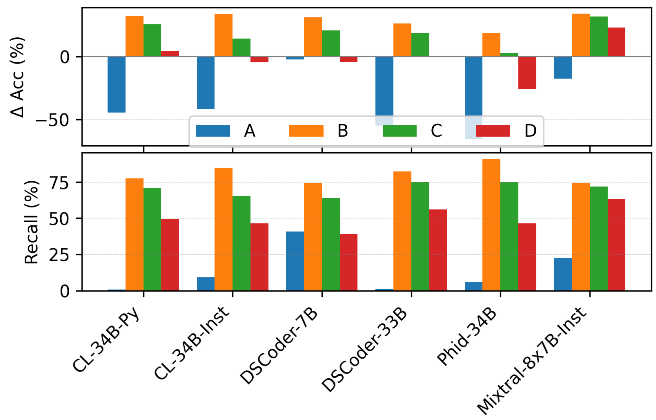

Figure 7 | **Task-Specific Accuracy and Performance Fluctuations Across Answer Options** Models exhibit marked fluctuations in accuracy depending on the position of the correct answer in Code Completion in CodeMMLU. Revealing the bias and inconsistencies in related coding multiple-choice question (MCQ) task and how sensitive LLMs are to answer ordering.

and real-world programming tasks. Our experiments highlighted the benchmark's key advantages, including its cost-effectiveness, scalability, and extensive task coverage. The insights gained revealed a strong correlation between software knowledge and real-world task performance, demonstrating that models with deeper comprehension outperform those relying purely on probabilistic generation.

Additionally, CodeMMLU provides more accurate and detailed rankings of LLMs, particularly in open-source models, where significant reordering of performance was observed. The benchmark also revealed inconsistencies in model comprehension when compared to traditional evaluations like HumanEval, emphasizing the need for more robust benchmarks that go beyond simple code generation.

Limitations, and Future Work

Limitations. While CodeMMLU offers a broad and diverse evaluation, there are some limitations. First, the MCQ format, though effective at testing comprehension, might not fully capture creative aspects of code generation or models' ability to optimize code. Second, the current scope of languages and tasks could be expanded to include more specialized domains or additional programming languages to better assess models' versatility.

Future Work. Looking forward, we plan to release CodeMMLU as an open-source benchmark for the research community. This release will include the full dataset, along with tools for automated evaluation, allowing for widespread adoption and further improvements. Future updates will focus on adding more complex tasks, refining the balance between real-world scenarios and theoretical knowledge, and incorporating user feedback to make the benchmark even more robust for next-generation LLMs.

Table 4 | **Performance Comparison between HumanEval and MCQ Code Completion Tasks.** The performance fluctuation highlights the selection biases observed when the correct (golden) answer is moved to positions A, B, C, or D.

Models	HumanEval	Code Completion MCQ A	B	C	D
CodeLlama-7B-Python	40.48	0.00 (-40.48)	90.24 (+49.76)	14.02 (-26.46)	0.61 (-39.87)
CodeLlama-7B-Instruct	45.65	3.66 (-41.99)	1.22 (-44.43)	93.90 (+48.25)	15.85 (-29.80)
CodeLlama-13B-Python	42.89	0.61 (-42.28)	54.88 (+11.99)	70.12 (+27.23)	12.20 (-30.69)
CodeLlama-13B-Instruct	50.6	2.44 (-48.16)	68.29 (+17.69)	72.56 (+21.96)	29.88 (-20.72)
CodeLlama-34B-Python	45.11	0.61 (-44.50)	77.44 (+32.33)	70.73 (+25.62)	49.39 4.28
CodeLlama-34B-Instruct	50.79	9.15 (-41.64)	84.76 (+33.97)	65.24 (+14.45)	46.34 (-4.45)
Deepseek-Coder-7B-base-v1.5	43.2	40.85 (-2.35)	74.39 (+31.19)	64.02 (+20.82)	39.02 (-4.18)
DeepSeek-Coder-33B-base	56.1	1.22 (-54.88)	82.32 (+26.22)	75.00 (+18.90)	56.10 (0.00)
Phind-CodeLLama-34B-v2	71.95	6.10 (-65.85)	90.85 (+18.90)	75.00 (+3.05)	46.34 (-25.61)
Mixtral-8x7B-Instruct-v0.1	40.2	22.56 (-17.64)	74.39 (+34.19)	71.95 (+31.75)	63.41 (+23.21)

References

The claude 3 model family: Opus, sonnet, haiku. URL https://api.semanticscholar.org/CorpusID:268232499.

Qwen2 technical report. 2024.

01. AI, :, Alex Young, Bei Chen, Chao Li, Chengen Huang, Ge Zhang, Guanwei Zhang, Heng Li, Jiangcheng Zhu, Jianqun Chen, Jing Chang, Kaidong Yu, Peng Liu, Qiang Liu, Shawn Yue, Senbin Yang, Shiming Yang, Tao Yu, Wen Xie, Wenhao Huang, Xiaohui Hu, Xiaoyi Ren, Xinyao Niu, Pengcheng Nie, Yuchi Xu, Yudong Liu, Yue Wang, Yuxuan Cai, Zhenyu Gu, Zhiyuan Liu, and Zonghong Dai. Yi: Open foundation models by 01.ai, 2024. URL https://arxiv.org/abs/2403.04652.

AI@Meta. Llama 3 model card. 2024. URL https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md.

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. Santacoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988*, 2023.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang

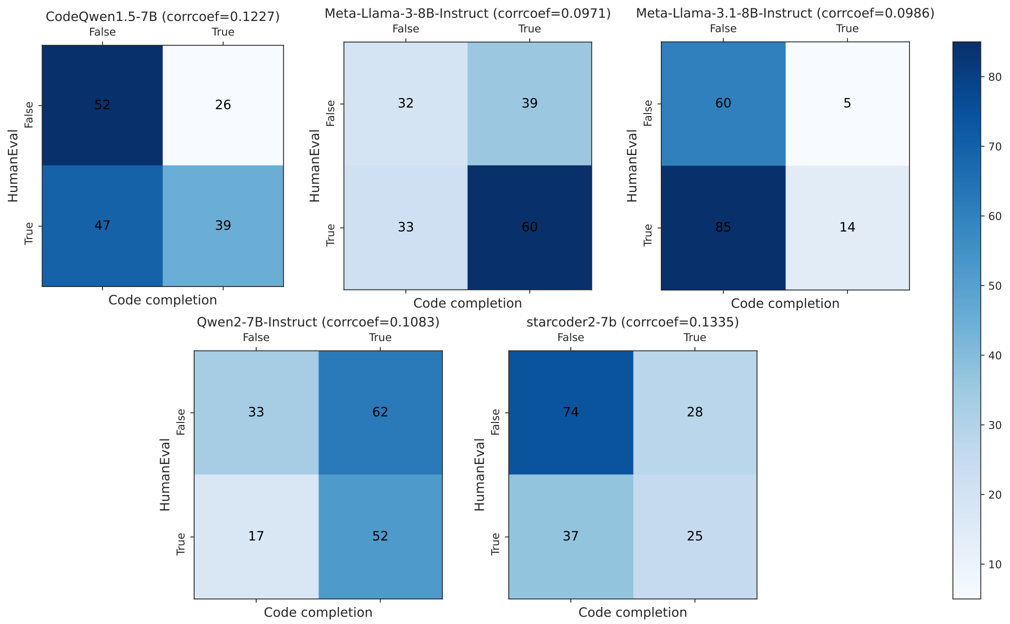

Figure 8 | **Comparison of CodeMMLU's code completion task and HumanEval.** Many LLMs show a performance discrepancy between the two tasks, where models that successfully passed the HumanEval code generation test often failed to select the correct answer in the multiple-choice (MCQ) format, or vice versa, for the same question.

Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.

Nghi DQ Bui, Hung Le, Yue Wang, Junnan Li, Akhilesh Deepak Gotmare, and Steven CH Hoi. Codetf: One-stop transformer library for state-of-the-art code llm. *arXiv preprint arXiv:2306.00029*, 2023.

Nicholas Carlini, Daphne Ippolito, Matthew Jagielski, Katherine Lee, Florian Tramer, and Chiyuan Zhang. Quantifying memorization across neural language models. *arXiv preprint arXiv:2202.07646*, 2022.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Hao Zhang, Banghua Zhu, Michael Jordan, Joseph E. Gonzalez, and Ion Stoica. Chatbot arena: An open platform for evaluating llms by human preference, 2024. URL https://arxiv.org/abs/2403.04132.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (eds.), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Olivier Duchenne, Onur Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaoqing Ellen Tan, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aaron Grattafiori, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alex Vaughan, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Franco, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, Danny Wyatt, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Erik Brinkman, Esteban

Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Firat Ozgenel, Francesco Caggioni, Francisco Guzmán, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Govind Thattai, Grant Herman, Grigory Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Igor Molybog, Igor Tufanov, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kai Wu, Kam Hou U, Karan Saxena, Karthik Prasad, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kun Huang, Kunal Chawla, Kushal Lakhotia, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Maria Tsimpoukelli, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikolay Pavlovich Laptev, Ning Dong, Ning Zhang, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Rohan Maheswari, Russ Howes, Ruty Rinott, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shiva Shankar, Shuqiang Zhang, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Sungmin Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Kohler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xiaocheng Tang, Xiaofang Wang, Xiaojian Wu, Xiaolan Wang, Xide Xia, Xilun Wu, Xinbo Gao, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang, Yuchen Hao, Yundi Qian, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, and Zhiwei Zhao. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

Lingyue Fu, Huacan Chai, Shuang Luo, Kounianhua Du, Weiming Zhang, Longteng Fan, Jiayi Lei, Renting Rui, Jianghao Lin, Yuchen Fang, Yifan Liu, Jingkuan Wang, Siyuan Qi, Kangning Zhang, Weinan Zhang, and Yong Yu. Codeapex: A bilingual programming evaluation benchmark for large language models. *ArXiv*, abs/2309.01940, 2023. URL https://api.semanticscholar.org/CorpusID:261530384.

GeeksforGeeks. GeeksforGeeks | Quiz Hub: Test Your Knowledge, 2024. URL https://www.geeksforgeeks.org/quizzes/.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I Wang. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*, 2024a.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I. Wang. Cruxeval: A benchmark for code reasoning, understanding and execution, 2024b. URL https://arxiv.org/abs/2401.03065.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

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

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

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

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mixtral of experts, 2024. URL https://arxiv.org/abs/2401.04088.

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

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020. URL https://arxiv.org/abs/2001.08361.

Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. xcodeeval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. *arXiv preprint arXiv:2303.03004*, 2023.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pp. 18319–18345. PMLR, 2023.

Linyi Li, Shijie Geng, Zhenwen Li, Yibo He, Hao Yu, Ziyue Hua, Guanghan Ning, Siwei Wang, Tao Xie, and Hongxia Yang. Inficoder-eval: Systematically evaluating the question-answering capabilities of code large language models. *arXiv preprint arXiv:2404.07940*, 2024.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. Quixbugs: a multi-lingual program repair benchmark set based on the quixey challenge. *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, 2017. URL https://api.semanticscholar.org/CorpusID:7158771.

Stephanie Lin, Jacob Hilton, and Owain Evans. Truthfulqa: Measuring how models mimic human falsehoods, 2022. URL https://arxiv.org/abs/2109.07958.

Chenxiao Liu and Xiaojun Wan. Codeqa: A question answering dataset for source code comprehension. *arXiv preprint arXiv:2109.08365*, 2021.

Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, and Li Zhang. Exploring and evaluating hallucinations in llm-powered code generation. *arXiv preprint arXiv:2404.00971*, 2024a.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36, 2024b.

Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. *arXiv preprint arXiv:2306.03091*, 2023.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder 2 and the stack v2: The next generation, 2024a. URL https://arxiv.org/abs/2402.19173.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*, 2024b.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*, 2023.

Alexandre Matton, Tom Sherborne, Dennis Aumiller, Elena Tommasone, Milad Alizadeh, Jingyi He, Raymond Ma, Maxime Voisin, Ellen Gilsenan-McMahon, and Matthias Gallé. On leakage of code generation evaluation datasets. *arXiv preprint arXiv:2407.07565*, 2024.

Milad Nasr, Nicholas Carlini, Jonathan Hayase, Matthew Jagielski, A Feder Cooper, Daphne Ippolito, Christopher A Choquette-Choo, Eric Wallace, Florian Tramèr, and Katherine Lee. Scalable extraction of training data from (production) language models. *arXiv preprint arXiv:2311.17035*, 2023.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea

Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. Gpt-4 technical report, 2024. URL https://arxiv.org/abs/2303.08774.

Yicheng Ouyang, Jun Yang, and Lingming Zhang. Benchmarking automated program repair: An extensive study on both real-world and artificial bugs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 440–452, 2024.

Pouya Pezeshkpour and Estevam Hruschka. Large language models sensitivity to the order of options in multiple-choice questions. pp. 2006–2017, 01 2024. doi: 10.18653/v1/2024.findings-naacl.130.

Nikhil Pinnaparaju, Reshinth Adithyan, Duy Phung, Jonathan Tow, James Baicoianu, Ashish Datta, Maksym Zhuravinskyi, Dakota Mahan, Marco Bellagente, Carlos Riquelme, et al. Stable code technical report. *arXiv preprint arXiv:2404.01226*, 2024.

Ruchir Puri, David Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian T Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. In J. Vanschoren and S. Yeung (eds.), *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1, 2021. URL https://datasets-benchmarks-proceedings.neurips.cc/paper_files/paper/2021/file/a5bfc9e07964f8dddeb95fc584cd965d-Paper-round2.pdf.

Mirza Masfiqur Rahman and Ashish Kundu. Code hallucination. *arXiv preprint arXiv:2407.04831*, 2024.

Vyas Raina, Adian Liusie, and Mark Gales. Is llm-as-a-judge robust? investigating universal adversarial attacks on zero-shot llm assessment. *arXiv preprint arXiv:2402.14016*, 2024.

Joshua Robinson, Christopher Michael Rytting, and David Wingate. Leveraging large language models for multiple choice question answering, 2023. URL https://arxiv.org/abs/2210.12353.

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

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024. URL https://arxiv.org/abs/2308.12950.

SanFoundry. SanFoundry, 2024. URL https://www.sanfoundry.com/.

Ben Shneiderman and Richard Mayer. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Parallel Programming*, 8:219–238, 06 1979. doi: 10.1007/BF00977789.

Zayne Sprague, Fangcong Yin, Juan Diego Rodriguez, Dongwei Jiang, Manya Wadhwa, Prasann Singhal, Xinyu Zhao, Xi Ye, Kyle Mahowald, and Greg Durrett. To cot or not to cot? chain-of-thought helps mainly on math and symbolic reasoning, 2024. URL https://arxiv.org/abs/2409.12183.

M.-A. Storey. Theories, methods and tools in program comprehension: past, present and future. In *13th International Workshop on Program Comprehension (IWPC'05)*, pp. 181–191, 2005. doi: 10.1109/WPC.2005.38.

Alon Talmor, Jonathan Herzig, Nicholas Lourie, and Jonathan Berant. Commonsenseqa: A question answering challenge targeting commonsense knowledge, 2019. URL https://arxiv.org/abs/1811.00937.

Hung Quoc To, Nghi DQ Bui, Jin Guo, and Tien N Nguyen. Better language models of code through self-improvement. *arXiv preprint arXiv:2304.01228*, 2023.

W3Schools. W3Schools.com, 2024. URL https://www.w3schools.com/quiztest/.

Peiyi Wang, Lei Li, Liang Chen, Zefan Cai, Dawei Zhu, Binghuai Lin, Yunbo Cao, Qi Liu, Tianyu Liu, and Zhifang Sui. Large language models are not fair evaluators, 2023a. URL https://arxiv.org/abs/2305.17926.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*, 2023b.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023. URL https://arxiv.org/abs/2201.11903.

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 1482–1494. IEEE, 2023.

Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. Measuring program comprehension: A large-scale field study with professionals. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 584–584, 2018. doi: 10.1145/3180155.3182538.

Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pp. 1–10, 2022.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence?, 2019. URL https://arxiv.org/abs/1905.07830.

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

Chujie Zheng, Hao Zhou, Fandong Meng, Jie Zhou, and Minlie Huang. Large language models are not robust multiple choice selectors. In *International Conference on Learning Representations*, 2024a. URL https://openreview.net/forum?id=shr9PXz7T0.

Chujie Zheng, Hao Zhou, Fandong Meng, Jie Zhou, and Minlie Huang. Large language models are not robust multiple choice selectors, 2024b. URL https://arxiv.org/abs/2309.03882.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36:46595–46623, 2023.

Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. Opencodeinterpreter: Integrating code generation with execution and refinement. *arXiv preprint arXiv:2402.14658*, 2024c.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*, 2024.

A. Appendix

A. Appendix

A.1. Prompt comparison

We experimented on 48 LLM models from over 12 family with 3 different settings, namely: Zero-shot, Few-shot and Chain-of-Thought (CoT). The overall trend can be demonstrated through GPT-4o and MetaLlama 3 70B (Fig. 9), the SOTA models in closed-source and open-source families. The LLMs show a significant decrease in performance in the Chain-of-Thought prompt setting compare to few-shot and zero-shot settings. As concluded in Sprague et al. (2024) final result, CoT bring extra "thinking" step into the original task which only effective on task involved math or logic while the result on benchmark like MMLU are identical with and without CoT prompting. The decreasing that happened mostly in CodeLLM knowledge test set align with Sprague et al. (2024) conclusion of the inefficient of CoT for non-reasoning task.

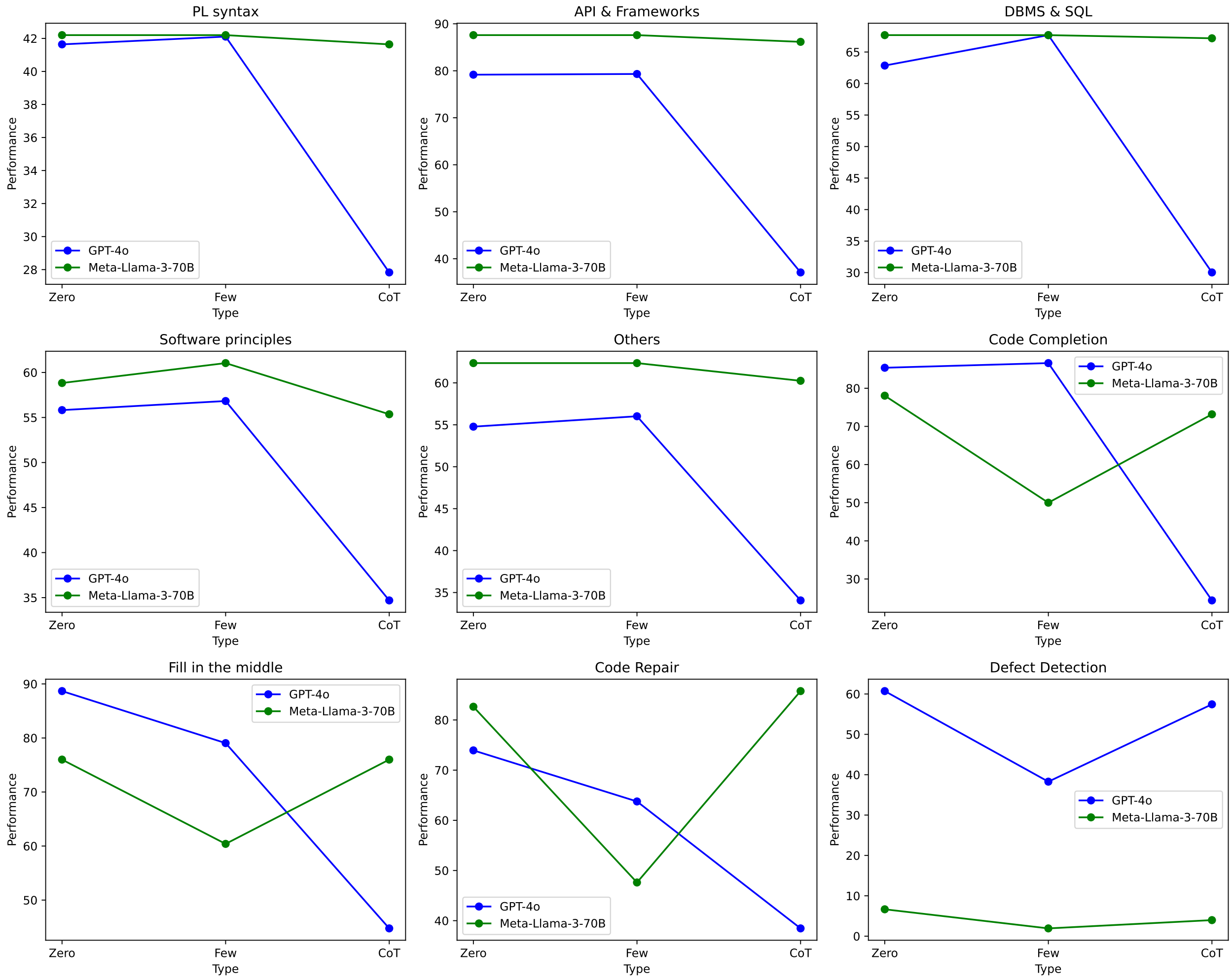

Figure 9 | **Comparison between GPT4o and Meta Llama-3 70B on various prompt settings**. We experiment with zero-shot, 1-shot, and CoT prompt configuration, where the result indicates the ineffectiveness of CoT in boosting the models' performance. Comparing to zeroshot config, 1-shot prompt slightly increase the performance in knowledge tasks but falls shorter in real tasks.

A.2. Data Cleaning and Filtering

A.2.1. Multiple-Choice Question (MCQ) Answering Filtering

The process of cleaning and filtering the MCQ data involved the following steps:

- **Incomplete Data**: Filtering instance with incomplete problem statement, code snippet, or solution.
- **Non-textual Content**: Filtering question containing images, videos, or other media. Yield dataset focused exclusively on text-based coding tasks.
- **Irrelevant or Redundant Content**: Any content not directly contributing to the assessment of programming knowledge or coding skills, such as off-topic questions or repeated instances, was filtered out.

A.2.2. Data quality Filtering

We maintain the integrity of the knowledge set and ensure it provides a reliable measure of LLM performance in coding and programming, we employed a rigorous data validation process, integrated deep-learning based filters, aimed at enhancing both the robustness and comprehensiveness of the dataset:

- **Common Error Identification**: We applied LLMs for scanning the dataset for common errors, such as ambiguous problem statements, incorrect solutions, or poorly constructed questions. Any instances that could lead to misleading assessments or misinterpretations were corrected or discarded.
- **Difficulty Level Assessment**: We targeted to remove all tasks that are deemed too easy for LLMs, as they do not significantly contribute to testing advanced problem-solving abilities. This result are achieved by benchmarking multiple LLMs to identify instances commonly answered correctly. Follow by a manual validated step to ensure their appropriateness for the benchmark.

Any data that failed to meet our stringent quality standards—whether due to errors, irrelevance, or inadequate difficulty—was filtered out, resulting in a high-quality, well-curated dataset tailored for evaluating LLMs on real-world coding challenges.

A.3. CodeMMLU Visualization

General knowledge MCQ example:

> The following are multiple-choice questions (with answers) about debugging a programming problem.
> **Question:** Suppose we have an O(n) time algorithm that finds the median of an unsorted array. Now consider a QuickSort implementation where we first find the median using the above algorithm, then use the median as a pivot. What will be the worst-case time complexity of this modified QuickSort?
> (A) $O(n^2 \log n)$
> (B) $O(n^2)$
> (C) $O(n \log n \log n)$
> (D) $O(n \log n)$

Code Completion example:

The following are multiple-choice questions (with answers) about programming problems.
Question: Which solution below is the most likely to complete the following code to achieve the desired goal?

```python
from typing import List

def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """ Check if in given list of numbers, are any two numbers closer to each
↪ other than given threshold.
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
    """
```

(A)

```python
    for i in range(len(numbers)):  # Change range to len(numbers)
        for j in range(i + 1, len(numbers)):
            if abs(numbers[i] - numbers[j]) < threshold:
                return True
        return False
```

(B)

```python
    return any(abs(a - b) < threshold for a, b \
        in zip(numbers, numbers[1:]))
```

(C)

```python
    for i in range(len(numbers) - 1):
        for j in range(i + 1, len(numbers)):
            if abs(numbers[i] - numbers[j]) > threshold:
                return False
        return True
```

(D)

```python
    for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(numbers):
            if idx != idx2:
                distance = abs(elem - elem2)
                if distance < threshold:
                    return True

    return False
```

Answer:

Fill in the blank example:

The following are multiple-choice questions (with answers) about a programming problem with incomplete solution.
Problem statement: You are given an array of intervals, where intervals[i] = [starti, endi] and each starti is unique. The right interval for an interval i is an interval j such that startj >= endi and startj is minimized. Note that i may equal j. Return an array of right interval indices for each interval i. If no right interval exists for interval i, then put -1 at index i.
Incomplete Solution:

```python
def find_right_interval(intervals):
```

```
    n = len(intervals)
    res = [-1] * n
    for i in range(n):
        intervals[i].append(i)

    def binary_search(ele):
        left, right = 0, n-1
        ans = float('inf')
        while left <= right:
            mid = (left + right) // 2
            if intervals[mid][0] >= ele:
                ans = min(ans, mid)
                right = mid - 1
            else:
                left = mid + 1
        return ans

    intervals.sort()
    for i in intervals:
        ------------------
        if val != float('inf'):
            res[i[2]] = intervals[val][2]
    return res
```

Question: The provided solution is missing a part, which option below is the most likely to complete the solution and achieve the desired goal?

(A)

```
    val = binary_search(i[1])
```

(B)

```
    if val != float('inf'):
```

(C)

```
    val = binary_search(i[1])
```

(D)

```
    if val != float('inf'):
      res[i[2]] = intervals[val][2]
```

Answer:

Code Repair example:

The following are multiple-choice questions (with answers) about debugging a programming problem.

Question: The following code snippet is producing incorrect results; Which solution below correctly identifies the bug and repairs it to achieve the desired goal?

```
import java.util.*;
public class DETECT_CYCLE {
    public static boolean detect_cycle(Node node) {
        Node hare = node;
        Node tortoise = node;
        while (true) {
            if (hare.getSuccessor() == null)
                return false;
            tortoise = tortoise.getSuccessor();
            hare = hare.getSuccessor().getSuccessor();
            if (hare == tortoise)
                return true;
        }
    }
```

```
15 }
```

(A) Modify line 6:

```
for (; ; ) {
```

(B) Modify line 7:

```
if (null==hare ||hare.getSuccessor() == null)
```

(C) Modify line 12:

```
return hare.getSuccessor() != null && hare == tortoise;
```

(D) Modify line 11:

```
if (Objects.equals(hare, tortoise))
```

Defect detection example:

The following are multiple-choice questions (with answers) about programming problem.
Question: Given a code snippet below, which behavior most likely to occur when running the solution?

```
import java.util.*;
public class Main {
    public static void main(string[] args) {
        Scanner sc = new Scanner(System.in);
        int A = sc.nextInt();
        int B = sc.nextInt();
        int T = sc.nextInt();
        int S = T/A System.out.println(s*b);
    }
}
```

(A) Memory Limit Exceeded
(B) Time Limit Exceeded
(C) Compilation Error
(D) The code not contain any issue

A.4. Prompt collection

A.4.1. Data creation prompt

Prompt used for synthesis distractor for real-world task:

Code Repair distractor creation prompts:

After extracting statement from buggy version, we use LLMs to rewrite a new version of that statement. We command LLMs to assume the bug is located in the assigned line and their target is correct that line. Here is the prompt:

```
Given a buggy Python code snippet, you will be asked to debugging the code.
```
def truncate_number(number: float) -> float:
    return number * (number % 1)
```
Let assume the bug is located in this line:
```    return number * (number % 1)```
Adjust this line in order to solve the bug.
The re-written line must be syntactic correct, executable and wrapped in ``` ```
↪ brace.
Don't give any details.
### Rewritten line:
```    return number % 1.0```

Given a buggy Java code snippet, you will be asked to debugging the code.
```{code}```
Let assume the bug is located in this line:
```{line}```
Adjust this line in order to solve the bug.
The re-written line must be syntactic correct, executable and wrapped in ``` ```
↪ brace.
Don't give any details.
### Rewritten line:
```

We executing the problem with given test cases. Our target is to create reasonable false answer that would require deep interpretation. Follow by an LLMs based filter to pick from pool of negative answer the most likely able to solve the buggy problem. This result a set of confusing negative answer. Those reasonable false sample with executable (and if they can pass through few testcases) is golden negative answer.

Fill in the blank distractor creation prompt:

From correct solution from leetcode, we randomly mask a line/a block of code and generate false answer (for multiple choice) from LLMs:

```
Following this code:
{code}
I prepare some multiple choice questions answering
so i want to make small change on this line
but it still look true of this line : {line}
help me generate 3 version change in this code and each output should in ``` ```
↪ brace and code only.
Don't give any details
```

A.4.2. Experiment prompt configuration

Zero-shot prompts

General knowledge MCQ test set:

```
        The following are multiple-choice questions (with answers) about software
↪ development.

        Question: {question}
        {multiple_choices}

        Answer:
```

Code completion:

```
        The following are multiple-choice questions (with answers) about software
↪ development.

        Question: {question}
        {multiple_choices}

        Answer:
```

Fill in the blank:

```
        The following are multiple-choice questions (with answers) about a programming
↪ problem with an incomplete solution.

        Problem statement: {question}

        Incomplete Solution:
        {codebase}

        Question: The provided solution is missing a part, Which option below is the most
↪ likely to complete the solution and achieve the desired goal?

        {multiple_choices}

        Answer:
```

Code Repair:

```
        The following are multiple-choice questions (with answers) about debugging a
↪ programming problem.

        Question: The implementation below is producing incorrect results. Which solution
↪ below correctly identifies the bug and repairs it to achieve the desired goal?
        {question}

        {multiple_choices}

        Answer:
```

Defect Detection:

```
        The following are multiple-choice questions (with answers) about programming
↪ problems.

        Question: Given a code snippet below, which behavior most likely to occur when
↪ execute it?
        {question}

        {multiple_choices}

        Answer:
```

Few-shot prompt

General knowledge MCQ test set:

```
        The following are multiple choice questions (with answers) about software
↪ development.

        Question: If a sorted array of integers is guaranteed to not contain duplicate
↪ values, in order to search a for a specific value which of the following algorithms is
↪ the most efficient for this task?

        (A) Bubble Sort (B) Linear Search (C) Insertion Sort (D) Binary Search

        Answer: The answer is (D).

        Question: {question}
        {multiple_choices}

        Answer:
```

Code completion:

```
        The following are multiple-choice questions (with answers) about programming
↪ problems.

        Question: Which solution below is the most likely completion the following code
↪ snippet to achieve the desired goal?
        ```python
        from typing import List

        def two_sum(nums: List[int], target: int) -> List[int]:
            """
            Given an array of integers nums and an integer target, return indices of the
↪ two numbers such that they add up to target.
            You may assume that each input would have exactly one solution, and you may not
↪ use the same element twice.

            >>> two_sum([2,7,11,15], 9)
            [0,1]
            >>> two_sum([3,2,4], 6)
            [1,2]
            >>> two_sum([3,3], 6)
            [0,1]
            """
        ```

        (A) ```python
            n = len(nums)
            for i in range(n - 1):
                for j in range(i + 1, n):
                    if nums[i] + nums[j] == target:
                        return [i, j]
            return []
        ```
        (B) ```python
            for num in nums:
                if target - num in nums:
                    return [nums.index(num), nums.index(target - num)]
            return []
        ```
```

```
(C) ```python
    for i in range(len(nums)):
        if nums[i] * 2 == target:
            return [i, i]
    return []
```
(D) ```python
    num_dict = {}
    for i, num in enumerate(nums):
        if target - num in num_dict:
            return [num_dict[target - num], i]
        num_dict[i] = num
    return []
```
Answer: The answer is A.

Question: Which solution below is the most likely completion the following code
↪ snippet to achieve the desired goal?
```python
{question}
```

{multiple_choices}

Answer:'''
```

Fill in the blank:

```
        The following are multiple-choice questions (with answers) about a programming
↪ problem with incomplete solution.

        Problem statement: You are given an array of intervals, where intervals[i] =
↪ [starti, endi] and each starti is unique.
        The right interval for an interval i is an interval j such that startj >= endi and
↪ startj is minimized.
        Note that i may equal j. Return an array of right interval indices for each
↪ interval i.
        If no right interval exists for interval i, then put -1 at index i.

        Incomplete Solution:
        python```
        def find_right_interval(intervals):
            n = len(intervals)
            res = [-1] * n
            for i in range(n):
                intervals[i].append(i)

            def binary_search(ele):
                left, right = 0, n-1
                ans = float('inf')
                while left <= right:
                    mid = (left + right) // 2
                    if intervals[mid][0] >= ele:
                        ans = min(ans, mid)
                        right = mid - 1
                    else:
                        left = mid + 1
                return ans

            intervals.sort()
            for i in intervals:
                _________________

            return res
        ```
        Question: The provided solution is missing a part, Which option below is the most
↪ likely to complete the solution and achieve the desired goal?
```
```

```
(A) ```python
    val = binary_search(i[1])
    if val != float('inf'):
        res[i[2]] = intervals[val][2]
```
(B) ```python
    if val != float('inf'):
        res[i[2]] = intervals[val][2]
    else:
        continue
```
(C) ```python
    val = binary_search(i[1])
    if val != float('inf'): res[i[2] + 1] = intervals[val][2]
```
(D) ```python
    if val != float('inf'):
        res[i[2]] = intervals[val][2]
    else:
      continue
```
Answer: The answer is (A).

Problem statement: {question}

Incomplete Solution:
{codebase}

Question: The provided solution is missing a part, Which option below is the most
↪ likely to complete the solution and achieve the desired goal?

{multiple_choices}

Answer:
```

Code Repair:

```
The following are multiple-choice questions (with answers) about debugging a
↪ programming problem.

Question: The implementation below is producing incorrect results.
Which solution below correctly identifies the bug and repairs it to achieve the
↪ desired goal?
{question}

{multiple_choices}

Answer:
```

Defect Detection:

```
        The following are multiple choice questions (with answers) about programming
↪ problem.

        Question: Given a code snippet below, which behavior most likely to occurr when
↪ execute it?
        ```python
        def chkPair(A, size, x):
            for i in range(0, size - 1):
                for j in range(i + 1, size):
                    if (A[i] + A[j] == x):
                        return 1
            return 0

        ```

        (A). The code contain no issue.
        (B). Memory Limit Exceeded
        (C). Internal error
        (D). Runtime Error

        Answer: The answer is (A).

        Question: Given a code snippet below, which behavior most likely to occurr when
↪ execute it?
        {question}

        {multiple_choices}

        Answer:
```

Chain-of-Thought zero-shot prompts

General knowledge MCQ test set:

```
        The following are multiple choice questions (with answers) about software devopment.

        Question: {question}
        {multiple_choices}

        Answer: Let's think step by step.
```

Code completion:

```
        The following are multiple choice questions (with answers) about programming
↪ problems.

        Question: Which solution below is the most likely completion the following code
↪ snippet to achieve the desired goal?
        ```python
        {question}
        ```
        {multiple_choices}

        Answer: Let's think step by step.
```

Fill in the blank:

```
        The following are multiple-choice questions (with answers) about a programming
↪ problem with uncomplete solution.

        Problem statement: {question}

        Incomplete Solution:
        {codebase}

        Question: The provided solution is missing a part, Which option below is the most
↪ likely to
        complete the solution and achieve the desired goal?

        {multiple_choices}

        Answer: Let's think step by step.
```

Code Repair:

```
        The following are multiple-choice questions (with answers) about debugging a
↪ programming problem.

        Question: The implementation below is producing incorrect results.
        Which solution below correctly identifies the bug and repairs it to achieve the
↪ desired goal?
        {question}

        {multiple_choices}

        Answer: Let's think step by step.
```

Defect Detection:

```
        The following are multiple-choice questions (with answers) about debugging a
↪ programming problem.

        The algorithm implementation below is producing incorrect results;
        Which solution below correctly identifies the bug and repairs it to achieve the
↪ desired goal?
        {question}

        {multiple_choices}

        Answer: Let's think step by step.
```

Chain-of-Thought few-shot prompts

General knowledge MCQ test set:

```
The following are multiple choice questions (with answers) about software devopment.

Question: If a sorted array of integers is guaranteed to not contain duplicate
↪ values, in order to search a for a specific value which of the following algorithms is
↪ the most efficient for this task?

(A) Bubble Sort (B) Linear Search (C) Insertion Sort (D) Binary Search

Answer: Let's think step by step. Binary Search is a divide-and-conquer algorithm
↪ that works by repeatedly dividing the search interval in half and searching for the
↪ value in the appropriate half. Since the array is already sorted and does not contain
↪ any duplicate value, this algorithm is optimal to find the desired value. The answer is
↪ (D).

Question: {question}
{multiple_choices}

Answer: Let's think step by step.
```

Code completion:

```
        The following are multiple choice questions (with answers) about programming
↪ problem.

        Question: Which solution below is the most likely completion the following code
↪ snippet to achieve the desired goal?
        ```python
        def is_vowel(char: str) -> bool:
            """
            Checks if the input character is a vowel.
            """
        ```

        (A) ```python
            return char.lower().is_vowel()
        ```
        (B) ```python
            vowels = set("aeiou")
            return char.lower() in vowels
        ```
        (C) ```python
            vowels = set("aeiou")
            return char.upper() in vowels
        ```
        (D) ```python
            vowels = "aeiou"
            return char.count(vowels) > 0
        ```

        Answer: Let's think step by step. The goal is to write a function is_vowel(char:
↪ str) -> bool that checks if the input character char is a vowel. The solution B
↪ correctly converts the input character to lowercase and checks if it is in the set of
↪ vowels.
        The answer is (B).

        Question: Which solution below is the most likely completion the following code
↪ snippet to achieve the desired goal?
        ```python
        {question}
        ```
        {multiple_choices}

        Answer: Let's think step by step.
```

Fill in the blank:

```
        The following are multiple-choice questions (with answers) about a programming
↪ problem with uncomplete solution.

        Problem statement: You are given an array of intervals, where intervals[i] =
↪ [starti, endi] and each starti is unique.
        The right interval for an interval i is an interval j such that startj >= endi and
↪ startj is minimized.
        Note that i may equal j. Return an array of right interval indices for each
↪ interval i.
        If no right interval exists for interval i, then put -1 at index i.

        Incomplete Solution:
        python```
        def find_right_interval(intervals):
            n = len(intervals)
            res = [-1] * n
            for i in range(n):
                intervals[i].append(i)
            def binary_search(ele):
                left, right = 0, n-1
                ans = float('inf')
                while left <= right:
                    mid = (left + right) // 2
                    if intervals[mid][0] >= ele:
                        ans = min(ans, mid)
                        right = mid - 1
                    else:
                        left = mid + 1
                return ans

            intervals.sort()
            for i in intervals:
                _________________

            return res
        ```


        Question: The provided solution is missing a part, Which option below is the most
↪ likely to
        complete the solution and achieve the desired goal?

        (A) ```python
            val = binary_search(i[1])
            if val != float('inf'):
                res[i[2]] = intervals[val][2]
        ```
        (B) ```python
            if val != float('inf'):
                res[i[2]] = intervals[val][2]
            else:
                continue
        ```
        (C) ```python
            val = binary_search(i[1])
            if val != float('inf'): res[i[2] + 1] = intervals[val][2]
        ```
        (D) ```python
            if val != float('inf'):
                res[i[2]] = intervals[val][2]
            else:
              continue
        ```
```
```

```
        Answer: Let's think step by step. The incomplete solution first sorts the intervals
↪ and then iterates over the sorted intervals. For each interval, it finds the right
↪ interval using a binary search.
        This option (A) finds the right interval index using the binary search and updates
↪ the result array accordingly.
        The option (B) is similar to (A), but it does not increment the index when finding
↪ the right interval index. This could lead to incorrect results.
        The option (C) increments the index when finding the right interval index. However,
↪ this is incorrect because the problem statement asks for the index of the right
↪ interval, not the offset from the original index.
        The option (D) uses the same index for both the original interval and the right
↪ interval, which could lead to incorrect results.
        The answer is (A).

        Problem statement: {question}

        Incomplete Solution:
        {codebase}

        Question: The provided solution is missing a part, Which option below is the most
↪ likely to
        complete the solution and achieve the desired goal?

        {multiple_choices}

        Answer: Let's think step by step.
```

Code Repair:

```
        The following are multiple-choice questions (with answers) about debugging a
↪ programming problem.

        Question: The implementation below is producing incorrect results.
        Which solution below correctly identifies the bug and repairs it to achieve the
↪ desired goal?
        {question}

        {multiple_choices}

        Answer: Let's think step by step.
```

Defect Detection:

```
        The following are multiple choice questions (with answers) about programming
↪ problem.

        Question: Given a code snippet below, which behavior most likely to occurr when
↪ execute it?
        ```python
        def chkPair(A, size, x):
            for i in range(0, size - 1):
                for j in range(i + 1, size):
                    if (A[i] + A[j] == x):
                        return 1
            return 0

        ```
        (A). The code contain no issue.
        (B). Memory Limit Exceeded
        (C). Internal error
        (D). Runtime Error

        Answer: Let's think step by step. The code appears to have no issues with typical
↪ valid inputs and will function as expected. It correctly checks for pairs of elements
↪ whose sum is x.
        The answer is (A).

        Question: Given a code snippet below, which behavior most likely to occurr when
↪ execute it?
        {question}
        {multiple_choices}

        Answer: Let's think step by step.
```