CoreCodeBench: A Configurable Multi-Scenario Repository-Level Benchmark

Lingyue Fu[1], **Hao Guan**[1], **Bolun Zhang**[1], **Haowei Yuan**[1], **Yaoming Zhu**[2], **Jun Xu**[3], **Zongyu Wang**[3], **Lin Qiu**[3], **Xunliang Cai**[3], **Xuezhi Cao**[3], **Weiwen Liu**[1], **Weinan Zhang**[1] **Yong Yu**[1]

[1]Shanghai Jiao Tong University, [2]AGI-EVAL, [3]Meituan

Abstract

As Large Language Models (LLMs) demonstrate increasingly sophisticated code processing capabilities, evaluating their performance on engineering-level code remains challenging. Existing repository-level benchmarks primarily focus on single scenarios, such as code generation or bug fixing, without adequately capturing the diversity and complexity of real-world software or project engineering workflows. Furthermore, these benchmarks suffer from limited controllability in question positioning and reliability issues in their generated test cases. To address these limitations, we present CorePipe, a fully automated pipeline that converts repositories into comprehensive benchmark test cases, and introduce CORECODEBENCH, a configurable multi-scenario repository-level benchmark. To simulate real engineering scenarios, CorePipe generates three types of atomic questions (Development, BugFix, and Test-Driven Development) specifically targeting core code segments. These atomic questions are further combined into three types of composite questions, with difficulty levels flexibly adjusted through hyperparameter tuning. CORECODEBENCH provides a comprehensive and extensive repository-level benchmark to investigate the applicability of LLMs in real-world engineering projects. Experiments with 16 LLMs across diverse scenarios reveal varying capabilities and offer multi-dimensional insights into LLM performance in engineering contexts. Code of CorePipe and data of CORECODEBENCH are available.

1 Introduction

With the continuous improvement in the code processing capabilities of Large Language Models (LLMs), more researchers are starting to focus on their applications in engineering-level code. Engineering-level code often involves complex dependencies and long-context interactions, posing unique challenges for LLMs. Specialized code LLMs, such as QwenCoder [16] and DeepSeek-Coder [13], have demonstrated exceptional programming capabilities in software engineering. LLM-based products such as Copilot, Windsurf, and Cursor, significantly reduce the complexity programmers face in engineering-level projects. As the code processing capabilities of LLMs continue to evolve, there is a growing need to systematically understand their strengths and limitations across different engineering scenarios. To assess the programming capabilities of these tools at an engineering level, it is crucial to establish an effective and fair evaluation standard.

Several benchmarks have been proposed, such as SWEBench [17], REPOEXEC [14], and BigCodeBench [38], to evaluate the ability of LLMs in implementing engineering-level code. These benchmarks are derived and refined from real-world repositories, ensuring a high degree of alignment with real engineering code development. They focus on tasks such as natural language to code

Submitted to 39th Conference on Neural Information Processing Systems (NeurIPS 2025). Do not distribute.

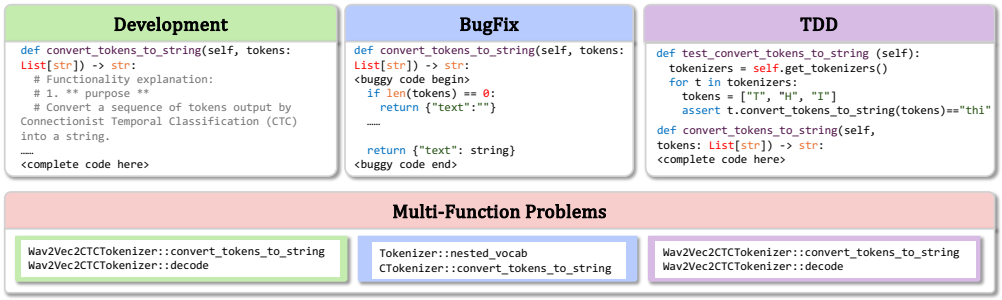

Figure 1: **Overview of CORECODEBENCH.**

(NL2Code) translation and bug fix within the scope of engineering code development. Although existing benchmarks provide an initial reference for evaluating the programming capabilities of LLMs in engineering environments, the current evaluation framework faces two critical challenges.

Challenge 1: Single Scenario. Prevailing repository-level benchmarks primarily focus on the code generation task, and do not adequately encompass the diverse scenarios present in engineering development. In real-world engineering practice, developers not only need to complete function-level code completion but also engage in bug fixies for unit tests. Additionally, within modular development paradigms, engineers often need to simultaneously implement main functions alongside supporting utility functions. These scenarios require the LLMs to display not only code generation capabilities but also cross-file contextual reasoning and implementation planning abilities—skills that current evaluation systems fail to systematically assess.

Challenge 2: Lack of Controllability and Reliability. Existing automated generation methods exhibit significant shortcomings in both controlling the positioning of generated questions and ensuring their reliability, directly impacting benchmark's effectiveness. The random masking approach, while achieving positional randomness, lacks logical constraints in mask selection, which might result in overlooking critical code segments or excessively testing non-essential areas [37]. Alternative approaches such as those based on cleaning pull requests, fix testing locations to historical revision points, limiting evaluation scenario diversity [17, 29]. These methods also suffer from low data reliability, with numerous pull requests not being self-contained and requiring substantial manual cleaning [28]. Neither method effectively ensures flexible control over test positioning while maintaining core code relevance and data quality, hindering comprehensive assessment of LLMs' performance in engineering-level tasks.

To address these limitations, we design a fully automated pipeline CorePipe that converts GitHub repositories into repository-level benchmark test cases. CorePipe generates three types of atomic questions (Development, Bugfix, Test-Driven Development) on core code segments, and further composes multiple composite question types with adjustable difficulty. Quality inspection and analysis show that the generated data are of high quality and reliability. As shown in Figure 1, we release a meticulously **Co**nfigurable **Re**pository-level benchmark, CORECODEBENCH, which effectively evaluates the actual capabilities and adaptability of LLMs in engineering-level code development. Through comprehensive evaluation of general-purpose and code-specific LLMs, we gain insights into the performance and characteristics of these models across diverse repository-level scenarios. CORECODEBENCH not only enables coarse-grained differentiation of LLM code abilities, but also provides fine-grained analysis of their potential. Flexible control of CorePipe over question difficulty enables CORECODEBENCH to offer a promising platform for future LLM evaluation. Our experiments further highlight several areas for improvement in LLMs' performance on engineering-level projects, paving the way for future advancements in model capabilities.

The contributions are summarized as follows:

- We design CorePipe, a fully automated pipeline for generating LLM engineering code capability tests from repository source code without any human intervention. CorePipe can be adapted to any programming language and any repository.
- We release the analysis and quality inspection results of the test data generated by CorePipe. The results demonstrate that CorePipe can produce high-quality and highly flexible test cases.

Benchmark	Multi-Task	Automatic	Difficulty Level	Flexible Position	Quality Inspection	Avg. Lines
SWEBench [17]	✗	✓	✗	✗	✗	38.01
DevBench [19]	✓	✗	✗	✗	✗	-
ExecRepoBench [37]	✗	✓	✗	✗	✗	2.42
Codev-Bench [29]	✗	✓	✗	✗	✗	43.69
EvoCodeBench [20]	✗	✗	✗	✗	✗	14.86
RepoMasterEval [34]	✗	✓	✗	✗	✗	-
BigCodeBench [38]	✗	✗	✗	✗	✗	13.55
REPOEXEC [14]	✗	✓	✗	✗	✓	21.9
CORECODEBENCH	✓	✓	✓	✓	✓	34.14

Table 1: Comparison between existing repository-level benchmarks and CORECODEBENCH.

- We provide CORECODEBENCH, a repository-level benchmark that includes three atomic tasks and three composite tasks. CORECODEBENCH features various question types and characteristics, offering new insights and analytical perspectives for evaluating LLM coding.
- We present the evaluation results on several state-of-the-art LLMs and conduct multifaceted analyses of their performance on repository-level scenarios.

2 Background and Related Work

2.1 Large Language Models for Code

General-purpose LLMs have demonstrated remarkable performance not only in natural language processing but also in code-related tasks. In recent years, LLMs tailored for code generation and reasoning have consistently achieved high scores in benchmark tests. On the HumanEval benchmark [7], the closed-source models Claude-3.5-Sonnet [3] and GPT-4o-0513 [24] have reached Pass@1 scores of 92.0% and 91.0%, respectively. Among open-source models, DeepSeek-Coder-V2-Instruct [9] and Qwen2.5-Coder-Instruct [16] have achieved Pass@1 scores of 90.2% and 88.4%. On other algorithmic problem benchmarks like MBPP [6], LLMs have surpassed Pass@1 scores of 85%, showcasing their strong performance in this domain. LLMs have also played a crucial role in engineering tasks, as demonstrated by products like Copilot [12], supporting code writing and debugging in extended context scenarios. To further advance coding LLMs, there is an urgent need for repository-level code benchmarks to evaluate performance in engineering contexts.

2.2 Existing Repository-level Benchmarks

Over the years, various benchmarks have been created to evaluate models on code-related tasks. Popular benchmarks focus on evaluating code generation (HumanEval [7], MBPP [6]), debugging (DebugBench [33], QuixBugs [15]), and code translation (CodeTransOcean [36]) capabilities. However, these benchmarks primarily target short code snippets and do not sufficiently address longer code generation or complex software engineering challenges.

Recently, with the enhanced code capabilities of LLMs and the support for larger context windows, several repository-level benchmarks have emerged. As demonstrated in Table 1, these benchmarks can automatically extract or generate test cases from real repositories to evaluate the performance of LLMs on repository-level code tasks. However, due to the random masking [37] or cleaning from pull requests [17, 29], the positioning, difficulty, and quality of the test cases are not consistently controlled. Some benchmarks [20, 38] require manual intervention to generate and validate test cases, thus preventing full automation. Furthermore, aside from DevBench [19], which evaluates LLMs' capabilities in software development through multi-stage tasks, most benchmarks [34, 14] have primarily concentrated on code generation within repository-level projects. Consequently, there is a clear need for a configurable, multi-scenario repository-level benchmark to fully assess the potential of LLMs in more complex software engineering contexts.

3 Method

In this section, we introduce the design of the CorePipe, including repository preprocessing, single-function problem generation, and multi-function problem generation. CorePipe is capable of identifying and rewriting core code segments to generate 6 types of problems, simulating various situations

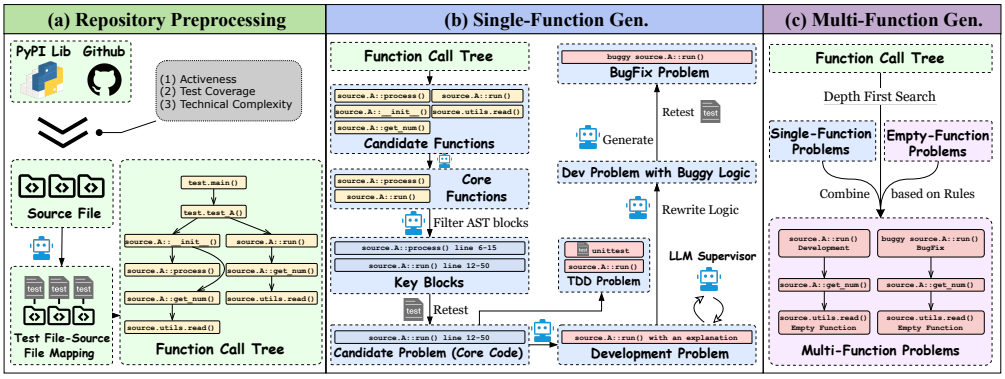

Figure 2: **Overview of CorePipe**. (a) *Repository Preprocessing* selects high-quality repositories based on three criteria, ensuring a diverse and representative codebase collection. (b) *Single-Function Problem Generation* creates three distinct types of problems focusing on individual function understanding and modification, targeting critical code segments. (c) *Multi-Function Problem Generation* constructs complex scenarios requiring an understanding of interactions between multiple functions.

in engineering development scenarios. For both single-function and multi-function problems, our pipeline ensures that the questions are generated from critical and representative locations, maintains the reliability of the generated problems, and allows for controllable difficulty levels.

3.1 Repository Preprocessing

Repository Selection. The PyPI library is a widely used public repository that offers a vast array of Python packages. We select open-source projects from PyPI based on the following criteria: (1) *Activeness*: the project has been updated or maintained within the past six months; (2) *Test Coverage*: the project contains unit tests, with test files accounting for more than 30% of the codebase; (3) *Technical Complexity*: the project has more than 5,000 lines of code and involves cross-module development. This selection process ensures that the chosen repositories not only reflect real-world engineering practices but also provide a solid testing infrastructure to support subsequent problem generation.

Test File-Source File Mapping Generation. We establish the mapping between source files and test files through a process that combines LLM-based analysis and automated rules. Specifically, we (1) use an LLM to analyze the repository's file tree structure; (2) apply automated rules to generate `<source, test>` pairs; and (3) perform executability checks and retain passing tests. The resulting mapping serves as a foundational data structure for subsequent problem generation, ensuring a strong semantic connection between test cases and target source code.

Function Call Tree Generation. For each validated test file and source file pair, we perform dynamic tracing on the test file to construct a cross-file function call tree. This process is implemented based on a customized version of the pycallgraph library [18]. Each node in the function call tree represents a function, annotated with its corresponding file and precise location. Every node serves as a potential candidate for Single-Function Problem generation, while the complete function call tree provides the structural foundation for composing multi-function Problems.

Prompts used in repository preprocessing stage is illustrated in Appendix A.

3.2 Single-Function Problems Generation

We first generate single-function problems as foundational atomic tasks, encompassing three types: Development, BugFix, and Test-Driven Development (TDD). These atomic tasks are designed to systematically evaluate the abilities of LLMs in long-context comprehension and local code implementation. Throughout the generation process, we dynamically monitor the quality of the questions, ultimately filtering out effective problems that meet the requirements of engineering practice.

Core Code Identification. Given that some functions in engineering code are simply basic condition checks or auxiliary utilities without core business logic, we first filter all function nodes in the function call tree to identify ***core functions*** as problem candidates. For each core function, we automatically select consecutive AST blocks as core code blocks by prompting LLMs to identify key segments, ensuring the completeness and centrality of the extracted segments. The retesting process verifies whether these core code blocks can be effectively detected by unit tests. All core functions and their associated core code blocks that pass the retesting process are considered as candidate problem locations.

Development Problem. We mask the identified core code blocks to generate development type problem. We then utilize the GPT-4o [24][1] to generate structured functional descriptions for the masked parts, ensuring that the descriptions cover key information such as input-output specifications, core logic, and boundary conditions. To further enhance the quality of the generated descriptions, we introduce Claude-3.5-Sonnet [4] as a discriminator model to score and provide feedback on the generated paragraphs. If deficiencies are detected, the generation model refines the descriptions based on the feedback. This iterative process is conducted twice. The specific prompt settings for this generation process are detailed in Appendix C.

BugFix Problem. Bug fixing is a common scenario faced by developers in real-world engineering projects. For current LLMs, the ability to fix syntactic errors is generally stronger than other error types [21]. Thus we focus more on constructing code snippets that contain logical errors. Specifically, we first use an LLM to rewrite development-oriented problems, generating erroneous logic descriptions for the masked code segments. Then, we employ a smaller-parameter LLM to produce buggy code for these masked segments. In our framework, large models are used to simulate more complex logical errors, while smaller models are used to generate more common and basic errors.

Test-Driven Development Problem. Test-Driven Development (TDD) is a software development approach where unit tests are written for target functionality before implementing the actual code. Following the methodology outlined in [22, 1], our TDD problems provide unit tests and require LLMs to implement the corresponding functionality based on these tests. TDD is a promising paradigm for helping ensure that the code generated by LLMs effectively captures the requirements. Specifically, we (1) select unit test code that directly tests specific functions based on the function call tree, (2) mask the core code block, (3) include the unit test code segments in the prompt. With the assistance of the function call tree, we ensure that the source code can be properly reconstructed using contextual information and the unit test.

3.3 Multi-Function Problem Generation

In engineering-level software projects, developers often extract parts of an implementation into separate utility functions for reuse. In such cases, a programmer may need to implement several sub-functions while developing a main function. Similarly, during bug fixing, it is sometimes necessary to address bugs across multiple related functions simultaneously. To simulate these real-world scenarios, we design Multi-Function Problems. Each Multi-Function Problem consists of multiple atomic problems, where an atomic problem refers to a single function that needs to be completed or corrected. Atomic problems include four types: development, BugFix, TDD, and empty-function. The Development, BugFix and TDD atomic problems are generated during the single-problem generation stage. For empty-function problems, the contents of utility functions in the repository are removed, leaving only the function signature and declaration. Empty-function problems are used exclusively within multi-function problems.

Each atomic problem corresponds to a node in the function call tree. The combination of atomic problems follows four basic rules: (1) at least one single-function problem is included; (2) the corresponding functions must have a call relationship (i.e., a parent-child relationship in the function call tree); (3) the maximum depth of the call tree is limited to d, where d is a hyperparameter; (4) the total number of atomic problems n satisfies $2 \leq n \leq \nu$, with ν as another hyperparameter. By adjusting the hyperparameters d and ν, we can control the complexity and difficulty of the generated problems. Specific generation rules for different subtypes are provided in Appendix D.

[1] Analysis of model selection for data generation is provided in Appendix B.

4 CoreCodeBench

4.1 Data Statistics

CORECODEBENCH encompasses a diverse collection of 12 repositories covering 6 distinct repository-level coding tasks, with a total of 1,545 valid problems. Detailed information about the repositories and illustration of CORECODEBENCH can be found in the Appendix E and F. In Table 2, we present the key statistics of CORECODEBENCH, including the average number of functions, average lines of gold solutions, and the number of problems for each problem type. The dataset encompasses a diverse range of problem complexities across different categories.

Problem Type	# Function	# Lines	# Problem
Development	1	17	422
BugFix	1	38	433
TDD	1	14	276
Multi-Dev.	3.85	53.92	167
Multi-BugFix	2.0	62.34	10
Mult-TDD	4.07	67.3	152
Difficult	4.75	65.66	91

Table 2: Data Statistics of CORECODEBENCH.

Each problem type contains specific contextual information to facilitate solution generation. Development problems include explanations of the masked code segments along with surrounding file context. BugFix problems contain the buggy code implementation, contextual information, and optional unit test details to aid in identifying and resolving errors. TDD problems provide file context and unit test code that defines the expected behavior of the implementation. For Multi-Function problems, we include code snippets of all relevant functions from the function call tree, offering a comprehensive view of the interdependent components. Examples of prompts for different problem types are presented in Appendix G.

4.2 Evaluation Metric

We assess the quality of generated code by executing unit tests corresponding to the source code. Following the method in [7], we adopt Pass@1 as our primary metric. For a given problem, Pass@1 indicates whether the first solution generated by a model successfully passes all associated unit tests. Additionally, we introduce PassRate as a complementary metric that measures the relative improvement over the retest baseline. PassRate is calculated as

$$\text{PassRate} = \frac{N_{\text{pass}} - N_{\text{retest}}}{N_{\text{total}} - N_{\text{retest}}},$$

where N_{pass} represents the number of test cases passed by the solution of model, N_{retest} is the number of test cases that pass without any modifications to the code, and N_{total} is the total number of test cases. While Pass@1 reflects the ability of a model to generate a fully correct solution in a single attempt, PassRate provides a finer-grained assessment by measuring the model's incremental improvement over the baseline, capturing partial correctness across all test cases.

For the overall CORECODEBENCH, both the Pass@1 score and PassRate are calculated as the average of their respective values across all repositories, providing a comprehensive measure of model performance across diverse codebases.

4.3 Quality Inspection

CorePipe utilizes an LLM supervisor to conduct preliminary quality assessment and filtering of generated problems. To further ensure problem quality, we implement additional quality inspection mechanisms specifically for Development-type problems.

IG Filter. For LLM-generated explanation texts, we introduce an Information Gain (IG) Score to measure the informational value provided by the explanations. Specifically,

$$\text{IG}_{\text{base}} = \text{PassRate}_{\text{exp}} - \text{PassRate}_{\text{no-exp}}$$

$\text{IG}_{\text{base}} > 0$ indicates that the explanation provides additional effective information, while $\text{IG}_{\text{base}} \leq 0$ suggests that the explanation information is redundant or incorrect. We select commonly used LLMs including GPT-4o [24], Claude-3.5-Sonnet [4], Doubao-pro-4k [11], and qwen-plus-latest [2] as baseline models. Based on the IG scores from these baseline LLMs, we retained only problems with $\text{IG}_{\text{base}} > 0$ and problems that none of the models could solve (i.e., difficult problems). After applying the IG filter, 48.56% of the problems are retained.

Single Function		Development		BugFix		TDD	
	Models	AC Rate	AC@1	AC Rate	AC@1	AC Rate	AC@1
API	GPT-4o [24]	82.09	57.47	57.95	34.42	84.09	46.38
	GPT-4.1 [26]	84.13	61.90	71.87	50.90	88.56	60.96
	o1-mini* [25]	76.85	47.02	57.28	32.68	78.92	54.74
	o4-mini (high)* [27]	86.66	59.29	69.51	50.65	87.13	70.21
	Claude-3.5-Sonnet [4]	86.83	61.41	63.80	40.47	85.88	60.56
	Claude-3.7-Sonnet* [5][2]	85.75	63.59	64.68	43.51	85.50	61.37
	Gemini-2.5-Pro-Preview [8]	73.21	48.06	30.79	22.67	74.50	51.60
	Grok-3* [35]	80.53	56.16	54.16	33.93	84.32	53.68
	Doubao-pro-4k [11]	76.25	43.54	63.19	39.43	76.10	31.24
	Doubao-1.5-pro [30]	84.22	57.70	64.69	41.43	83.26	45.50
	qwen-plus-latest [2][3]	78.82	52.96	39.91	22.05	80.96	40.02
	Qwen2.5-max [31]	83.06	57.85	50.87	28.18	82.83	47.65
Open-Source	DeepSeek-Coder-V2-Lite-Instruct-16B [9]	64.85	16.53	27.31	12.28	65.85	27.8
	DeepSeek-R1* [10]	84.58	58.81	66.48	45.07	79.23	56.66
	Llama3.1-70B [23]	71.53	41.00	51.93	28.64	79.42	37.33
	Qwen3-8B [32]	53.62	8.25	23.83	6.18	59.97	18.91

Table 3: Leaderboard of Single-Function Scenarios. Models using thinking mode are marked with *.

Manual Inspection. We further enlist experienced code engineers to annotate the problems. These annotators conducted quality checks on problems that had passed the IG filter. The quality assessment evaluated three aspects: readability, accuracy, and completeness, with flawed test cases being marked as unqualified. We randomly sampled 30 problems from each repository for inspection. Ultimately, the qualification rate for CORECODEBENCH (Development Problems) is 78.55%. This high qualification rate demonstrates that the problems originally generated by CorePipe are inherently reliable. Additionally, we have released the manually verified subset as `CoreCodeBench-Dev-Verified` alongside the main benchmark. We list the detailed experience of three human annotators in Appendix H, where all of them have a bachelor's degree in computer-related major, and at least 3 years of Python development experience.

5 Experiments

5.1 Setups

Models. We present a comprehensive evaluation of a diverse set of LLMs on our proposed CORECODEBENCH. The selected models represent a wide spectrum of architectures and parameter sizes, ranging from 7B to 70B parameters. Our evaluation covers both open-source models and proprietary API-based models released by leading AI research organizations. For models that support chain-of-thought (CoT) reasoning, we explicitly enable their reasoning capabilities during inference in order to fully assess their potential for complex reasoning tasks.

Implementation Details. All evaluations are performed using the officially recommended inference parameters for each model, including temperature, top_p, and top_k, whenever such recommendations are available. For models without specific recommendations, we employ deterministic sampling settings (temperature= 0, top_k= 1, top_p= 0.0) to ensure reproducible outputs. Other Implementation details specific to other question types are provided in Appendix I.

5.2 Main Result of Single-Function Problems

Table 3 presents the performance of various LLMs on the CORECODEBENCH-*Single* benchmark. We draw the following conclusions: (1) ***Model Performance***: Claude-3.7 and o4-mini (high) consistently achieve leading results across all three problem types, demonstrating the strong capabilities of recent proprietary models. Among open-source models, DeepSeek-R1 stands out with comparatively better results. Generally, models with larger parameter sizes outperform their smaller counterparts, and newer model versions exhibit clear advancements over previous generations, indicating continuous progress in model architecture and training techniques. (2) ***Metric Comparison***: The differing rankings produced by AC Rate and AC@1 indicate that these metrics provide complementary insights into model performance. AC@1 evaluates coarse-grained absolute performance, offering a clear

[2]Claude-3.7-Sonnet is a hybrid reasoning model.

[3]In this paper, we use qwen-plus-latest-2025-01-25.

Multi Function		Development		BugFix		TDD	
Models		AC Rate	AC@1	AC Rate	AC@1	AC Rate	AC@1
API	GPT-4o [24]	17.31	5.69	0.21	0	18.44	6.78
	GPT-4.1 [26]	12.85	3.77	44.00	20.00	22.22	8.11
	o1-mini* [25]	16.92	2.62	41.40	20.00	22.22	8.11
	o4-mini (high)* [27]	20.85	6.62	42.60	20.00	34.11	20.22
	Claude-3.5-Sonnet [4]	24.38	7.77	41.40	20.00	24.38	7.77
	Claude-3.7-Sonnet* [5]	35.54	13.85	41.60	20.00	31.56	17.11
	Gemini-2.5-Pro-Preview [8]	22.74	6.85	2.20	0	20.22	6.89
	Grok-3* [35]	25.62	14.46	15.40	0	15.44	7.44
	Doubao-pro-4k [11]	3.85	0	19.80	0	3.00	1.56
	Doubao-1.5-pro [30]	3.08	0	36.40	20.00	0.22	0
	qwen-plus-latest [2]	21.31	8.00	27.60	0	19.22	6.89
	Qwen2.5-max [31]	23.46	9.31	49.20	40.00	23.89	8.22
Open-Source	DeepSeek-Coder-V2-Lite-Instruct-16B [9]	0	0	0	0	1.22	1.22
	DeepSeek-R1* [10]	20.23	5.54	22.40	0	23.56	9.56
	Llama3.1-70B [23]	19.00	4.92	37.60	20.00	19.44	6.56
	Qwen3-8B [32]	0	0	13.8	0	1.78	1.22

Table 4: Leaderboard of Multi-Function Scenarios. Models using thinking mode are marked with *.

stratification of code generation capabilities among models. In contrast, AC Rate is able to capture performance differences within the same tier, serving as a finer-grained indicator of a model's potential to pass individual test cases. (3) ***Task Comparison***: The relatively lower scores in the BugFix scenario across all models highlight the increased complexity and difficulty of debugging tasks, suggesting valuable directions for future model improvement and research. More detailed results and repository-level breakdowns are provided in Appendix J.

5.3 Main Results of Multi-function Problems

Table 4 summarizes the performance of various models on the CORECODEBENCH-*Multi* benchmark. Compared to the single-function setting, scores for multi-function problems are significantly lower across all models and scenarios, highlighting the increased complexity and challenges posed by multi-function code generation tasks. Claude-3.7-Sonnet achieves the highest performance among all evaluated models, particularly excelling in the Development and TDD scenarios, which demonstrates its strong generalization and reasoning abilities in more complex contexts. Notably, in the BugFix scenario, due to stricter generation rules and a smaller number of available problems, the differences in AC@1 scores among models are less pronounced. However, AC Rate remains effective in distinguishing model performance, as it captures more granular improvements even when absolute success rates are low. More detailed results and repository-level breakdowns are provided in Appendix K.

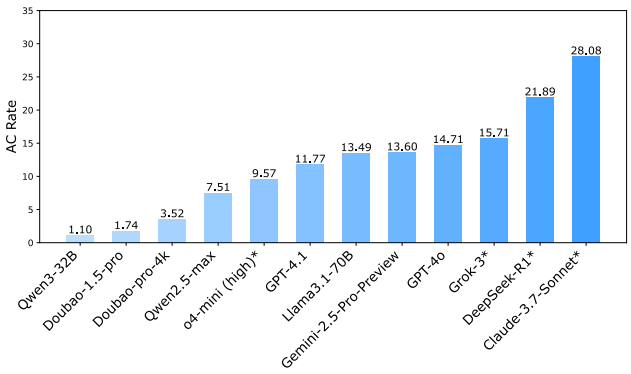

Figure 3: **CORECODEBENCH-*Difficult* Performance**.

In the multi-function scenario, models are required to provide completions for multiple functions within a single response (see Appendix G for prompt details). Ideally, an LLM would demonstrate planning in its implementation order, such as first completing simple utility functions and then implementing functions that invoke them, or vice versa–reflecting the diverse habits of human engineers.Our analysis reveals that, with the exception of DeepSeek16B-Coder-V2-Lite, most models tend to output answers strictly following the order of the functions as presented in the input prompt. This observation suggests that *current models lack flexible planning and hierarchical reasoning abilities when generating multi-function code*, often defaulting to a sequential approach rather than optimizing for logical or functional dependencies.

CORECODEBENCH-*Difficult* To further guide the development of future LLMs and to push the boundaries of current code generation capabilities, we introduce the CORECODEBENCH-*Difficult* dataset. Specifically, we generate this benchmark by setting the multi-problem generation hyperparameter $\nu = \infty$ (while keeping $d = 3$ to mimic real-world development environments). Figure 3 presents the AC Rate of various models on CORECODEBENCH-*Difficult*. Notably, the pass rates for all models remain below 30%, underscoring the substantial challenges posed by this dataset. These results highlight the effectiveness of the CORECODEBENCH-*Multi* benchmark in revealing the limitations of current models and providing a rigorous testbed for driving future advancements in code understanding and generation.

5.4 Coding Capabilities of LLMs

We claim that CORECODEBENCH enables comprehensive evaluation of multiple coding capabilities of LLMs. To visualize these capabilities, in Figure 4, we select nine representative model series and plot radar charts based on their performance across the six distinct scenarios defined in CORECODEBENCH. Each scenario is designed to assess a different aspect of coding ability, thus providing a multi-faceted view of model strengths and weaknesses. For clearer and more intuitive comparison, we normalize the results for each scenario, allowing us to better highlight the differences and relative rankings among models.

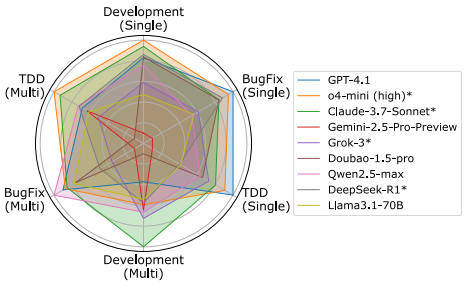

Figure 4: Performance of LLMs on CORECODEBENCH across scenarios.

Several key observations can be drawn from the radar charts. (1) The relative ranking of models differs across the six scenarios, indicating that CORECODEBENCH effectively evaluates multiple dimensions of LLMs' coding capabilities rather than a single aspect. (2) For Development and TDD problems, model performance in multi-function scenario does not always correlate with that in single-function scenario. This suggests that developing multiple interrelated functions requires additional abilities, such as deeper contextual understanding and implementation order planning. (3) For BugFix problems, model performance in single-function and multi-function scenarios is strongly correlated. This reflects the distinct nature of debugging tasks compared to development tasks, where debugging may rely more on local error correction skills that generalize across different granularities. Overall, these findings demonstrate the value of CORECODEBENCH as a multi-dimensional evaluation framework and highlight the necessity for continued research to develop LLMs with robust and versatile coding skills.

6 Conclusions & Limitations

In this paper, we present CorePipe, a fully automated pipeline for generating high-quality, diverse, and controllable repository-level benchmark test cases, and introduce CORECODEBENCH, a configurable benchmark that comprehensively evaluates LLMs' capabilities in real-world engineering scenarios. Through extensive experiments, we demonstrate that CORECODEBENCH enables both coarse and fine grained analysis of LLMs' coding abilities, revealing significant performance differences across various tasks and highlighting areas where current models still fall short, especially in complex and multi-function engineering contexts. Our work provides a scalable and rigorous testbed for the systematic assessment and future improvement of LLMs in engineering-level code development, paving the way for more robust and adaptable AI-driven software engineering tools.

Despite the automated generation of six types of questions from GitHub repositories achieved by CorePipe, our pipeline currently relies on the presence of comprehensive unit tests within the repositories. Repositories lacking sufficient unit tests cannot be processed by our current framework. In future work, we plan to enhance CorePipe by incorporating techniques for generating or augmenting unit tests, thereby expanding its applicability to a broader range of projects. Additionally, CORECODEBENCH currently focuses exclusively on Python repositories. We aim to extend support to other major programming languages, such as Java and C++, to enable more comprehensive evaluation of engineering capabilities.

References

[1] Toufique Ahmed, Martin Hirzel, Rangeet Pan, Avraham Shinnar, and Saurabh Sinha. Tdd-bench verified: Can llms generate tests for issues before they get resolved?, 2024. URL https://arxiv.org/abs/2412.02883.

[2] aliyun. https://help.aliyun.com/zh/model-studio/what-is-qwen-llm, 2025. Accessed: 2025-04-10.

[3] Anthropic. The claude 3 model family: Opus, sonnet, haiku, 2024. URL https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf. Accessed: 2024-05-10.

[4] Anthropic. Introducing claude 3.5 sonnet. https://www.anthropic.com/news/claude-3-5-son-net, 2024. Accessed: 2024-06-21.

[5] Anthropic. Claude 3.7 sonnet and claude code. https://www.anthropic.com/news/claude-3-7-sonnet, 2025. Accessed: 2025-02-25.

[6] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021. URL https://arxiv.org/abs/2108.07732.

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

[8] Google DeepMind. Gemini. https://deepmind.google/technologies/gemini/.

[9] DeepSeek-AI, Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y. Wu, Yukun Li, Huazuo Gao, Shirong Ma, Wangding Zeng, Xiao Bi, Zihui Gu, Hanwei Xu, Damai Dai, Kai Dong, Liyue Zhang, Yishi Piao, Zhibin Gou, Zhenda Xie, Zhewen Hao, Bingxuan Wang, Junxiao Song, Deli Chen, Xin Xie, Kang Guan, Yuxiang You, Aixin Liu, Qiushi Du, Wenjun Gao, Xuan Lu, Qinyu Chen, Yaohui Wang, Chengqi Deng, Jiashi Li, Chenggang Zhao, Chong Ruan, Fuli Luo, and Wenfeng Liang. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence, 2024. URL https://arxiv.org/abs/2406.11931.

[10] DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun,

T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL https://arxiv.org/abs/2501.12948.

[11] Volc Engine. https://www.volcengine.com/docs/82379/1159178, 2025. Accessed: 2025-04-22.

[12] GitHub. Github copilot: Your ai pair programmer, 2023. URL https://github.com/features/copilot/.

[13] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024. URL https://arxiv.org/abs/2401.14196.

[14] Nam Le Hai, Dung Manh Nguyen, and Nghi D. Q. Bui. On the impacts of contexts on repository-level code generation, 2025. URL https://arxiv.org/abs/2406.11927.

[15] Haichuan Hu, Ye Shang, Guolin Xu, Congqing He, and Quanjun Zhang. Can gpt-o1 kill all bugs? an evaluation of gpt-family llms on quixbugs, 2024. URL https://arxiv.org/abs/2409.10033.

[16] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-coder technical report, 2024. URL https://arxiv.org/abs/2409.12186.

[17] 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?, 2024. URL https://arxiv.org/abs/2310.06770.

[18] Graham King. pycallgraph: Python call graph visualizer. https://github.com/gak/pycallgraph.

[19] Bowen Li, Wenhan Wu, Ziwei Tang, Lin Shi, John Yang, Jinyang Li, Shunyu Yao, Chen Qian, Binyuan Hui, Qicheng Zhang, Zhiyin Yu, He Du, Ping Yang, Dahua Lin, Chao Peng, and Kai Chen. Prompting large language models to tackle the full software development lifecycle: A case study, 2024. URL https://arxiv.org/abs/2403.08604.

[20] Jia Li, Ge Li, Xuanming Zhang, Yihong Dong, and Zhi Jin. Evocodebench: An evolving code generation benchmark aligned with real-world code repositories, 2024. URL https://arxiv.org/abs/2404.00599.

[21] Shukai Liu, Linzheng Chai, Jian Yang, Jiajun Shi, He Zhu, Liran Wang, Ke Jin, Wei Zhang, Hualei Zhu, Shuyue Guo, Tao Sun, Jiaheng Liu, Yunlong Duan, Yu Hao, Liqun Yang, Guanglin Niu, Ge Zhang, and Zhoujun Li. Mdeval: Massively multilingual code debugging, 2025. URL https://arxiv.org/abs/2411.02310.

[22] Noble Saji Mathews and Meiyappan Nagappan. Test-driven development and llm-based code generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, page 1583–1594. ACM, October 2024. doi: 10.1145/3691620.3695527. URL http://dx.doi.org/10.1145/3691620.3695527.

[23] Meta. Introducing llama 3.1: Our most capable models to date. https://ai.meta.com/blog/meta-llama-3-1/, 2024. Accessed: 2024-07-23.

[24] OpenAI. Hello gpt-4o. https://openai.com/index/hello-gpt-4o/, 2024. Accessed: 2024-05-10.

[25] OpenAI. Openai o1-mini: Advancing cost-efficient reasoning. https://openai.com/index/openai-o1-mini-advancing-cost-efficient-reasoning/, 2024. Accessed: 2024-09-12.

[26] OpenAI. Introducing gpt-4.1 in the api. https://openai.com/index/gpt-4-1/, 2025. Accessed: 2025-04-14.

[27] OpenAI. Introducing openai o3 and o4-mini. https://openai.com/index/introducing-o3-and-o4-mini/, 2025. Accessed: 2025-04-16.

[28] OpenAI Team. Introducing SWE-Bench verified. https://openai.com/index/introducing-swe-bench-verified/, 2024. Accessed: August 13, 2024.

[29] Zhenyu Pan, Rongyu Cao, Yongchang Cao, Yingwei Ma, Binhua Li, Fei Huang, Han Liu, and Yongbin Li. Codev-bench: How do llms understand developer-centric code completion?, 2024. URL https://arxiv.org/abs/2410.01353.

[30] Doubao Team. Doubao 1.5pro. https://seed.bytedance.com/en/special/doubao_1_5_pro, 2025. Accessed: 2025-01-22.

[31] Qwen Team. Qwen2.5-max: Exploring the intelligence of large-scale moe model. https://qwenlm.github.io/blog/qwen2.5-max/, 2025. Accessed: 2025-01-28.

[32] Qwen Team. Qwen3: Think deeper, act faster. https://qwenlm.github.io/blog/qwen3/, 2025. Accessed: 2025-04-29.

[33] Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, Yesai Wu, Haotian Hui, Weichuan Liu, Zhiyuan Liu, and Maosong Sun. Debugbench: Evaluating debugging capability of large language models, 2024. URL https://arxiv.org/abs/2401.04621.

[34] Qinyun Wu, Chao Peng, Pengfei Gao, Ruida Hu, Haoyu Gan, Bo Jiang, Jinhe Tang, Zhiwen Deng, Zhanming Guan, Cuiyun Gao, Xia Liu, and Ping Yang. Repomastereval: Evaluating code completion via real-world repositories, 2024. URL https://arxiv.org/abs/2408.03519.

[35] x.AI. Grok 3 beta — the age of reasoning agents. https://x.ai/news/grok-3/, 2025. Accessed: 2025-02-19.

[36] Weixiang Yan, Yuchen Tian, Yunzhe Li, Qian Chen, and Wen Wang. Codetransocean: A comprehensive multilingual benchmark for code translation, 2023. URL https://arxiv.org/abs/2310.04951.

[37] Jian Yang, Jiajun Zhang, Jiaxi Yang, Ke Jin, Lei Zhang, Qiyao Peng, Ken Deng, Yibo Miao, Tianyu Liu, Zeyu Cui, Binyuan Hui, and Junyang Lin. Execrepobench: Multi-level executable code completion evaluation, 2024. URL https://arxiv.org/abs/2412.11990.

[38] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, Thong Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, Binyuan Hui, Niklas Muennighoff, David Lo, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro Von Werra. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions, 2025. URL https://arxiv.org/abs/2406.15877.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The paper proposes an automated pipeline for generating LLM engineering code, which is reflected in the abstract and introduction.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We discuss the limitations of the work in Section 6.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA]

Justification: There are no theoretical results in the paper.

Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: All essential details regarding data generation, experimental setup, evaluation metrics, and hyperparameters are fully described in the Section 5.1 and Appendix I, ensuring reproducibility of the main results.

Guidelines:

- The answer NA means that the paper does not include experiments.
- If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: We open-source data in Huggingface (CORECODEBENCH-*Single*, CORECODEBENCH-*Multi*).

Guidelines:

- The answer NA means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

Answer: [Yes]

Justification: We list the details in Section 5.1

Guidelines:

- The answer NA means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer:[No]

Justification: We do not perform statistical significance tests, as our conclusions focus on overall performance trends rather than asserting significant superiority of one model over another.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.

- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer:[Yes]

   Justification: We mainly rely on API-based models, and provide the details in 5.1.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The data is constructed under NeurIPS Code of Ethics.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [NA]

    Justification: As a coding benchmark and pipeline, the paper has no societal impacts as we might expect.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.

- If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: As a coding benchmark and pipeline, the paper has no such risk as we might expect.

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: We credit all models used in the paper via citation.

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.

- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: The assets is included in the HuggingFace/GitHub Repo.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [Yes]

    Justification: We provide human annotators with detailed task instructions, inform them of data open-source plans, and compensate them in accordance with local labor regulations.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: The crowdsourcing do not study human as subjects in the paper.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.

- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

    Answer: [NA]

    Justification: We use LLM only for writing, editing, or formatting purposes.

    Guidelines:

    - The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.