CoreCodeBench: Decoupling Code Intelligence via Fine-Grained Repository-Level Tasks

Anonymous ACL submission

Abstract

The evaluation of Large Language Models (LLMs) for software engineering has shifted towards complex, repository-level tasks. However, existing benchmarks predominantly rely on coarse-grained pass rates that treat programming proficiency as a monolithic capability, obscuring specific cognitive bottlenecks. Furthermore, the static nature of these benchmarks renders them vulnerable to data contamination and performance saturation. To address these limitations, we introduce CoreCodeBench, a configurable repository-level benchmark designed to dissect coding capabilities through atomized tasks. Leveraging our automated framework, CorePipe, we extract and transform Python repositories into a comprehensive suite of tasks that isolate distinct cognitive demands within identical code contexts. Unlike static evaluations, CoreCodeBench supports controllable difficulty scaling to prevent saturation and ensures superior data quality. It achieves a 78.55% validity yield, significantly surpassing the 31.7% retention rate of SWE-bench-Verified. Extensive experiments with state-of-the-art LLMs reveal a significant capability misalignment, evidenced by distinct ranking shifts across cognitive dimensions. This indicates that coding proficiency is non-monolithic, as strength in one aspect does not necessarily translate to others. These findings underscore the necessity of our fine-grained taxonomy in diagnosing model deficiencies and offer a sustainable, rigorous framework for evolving code intelligence. Code of CorePipe framework and data of CoreCodeBench are available in https://anonymous.4open.science/r/CoreCodeBench-64E2/.

1 Introduction

The evaluation of Large Language Models (LLMs) for code has evolved from function-level snippets (Chen et al., 2021; Austin et al., 2021) to complex, repository-level software engineering tasks, including code generation (Yang et al., 2024), program repair (OpenAI, 2024) and unit test generation (Huang et al., 2025). While these pioneering benchmarks (Zhuo et al., 2025; Hai et al., 2025; Xu et al., 2025) simulate realistic workflows, they predominantly rely on coarse-grained pass rates, thereby treating programming proficiency as a monolithic capability.

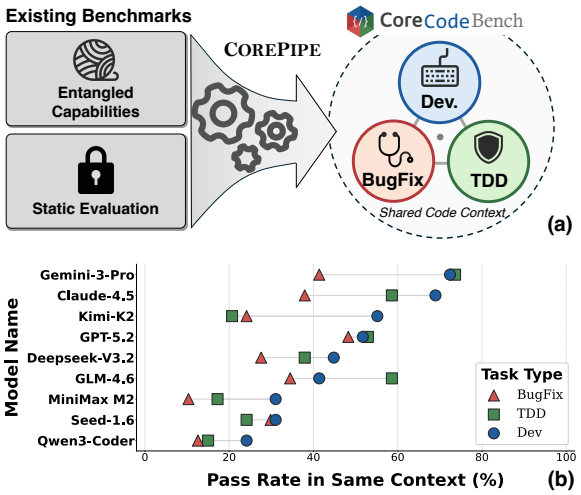


Figure 1: **Decomposing Code Intelligence.** (a) CoreCodeBench isolates distinct cognitive demands (Dev, BugFix, TDD) within an identical code context. (b) Performance comparison across these dimensions reveals significant **capability misalignment**, highlighting that coding proficiency is non-monolithic.

However, this coarse-grained approach obscures the distinct cognitive demands inherent to different engineering scenarios. Real-world software engineering entails switching between generating code from intent (Development), reasoning about logic errors (BugFix), and planning verification constraints (Test Driven Development, TDD). By conflating these dimensions under a single metric, existing benchmarks mask specific cognitive bottlenecks. As illustrated in Figure 1(b), when evaluating models on these specialized tasks derived from an identical code context, we observe significant **capability misalignment**. Contrary to the

expectation of uniform proficiency, models exhibit distinct ranking shifts across these dimensions. For instance, Kimi-K2 excels in Development but falters in BugFix and TDD, with a pass rate disparity exceeding 35%. This dimensional inconsistency underscores that coding proficiency is not monolithic—high generative performance does not guarantee the grounded reasoning required for robust software engineering.

Compounding the granularity issue is the **static nature** of existing benchmarks, which limits their longevity and validity. By representing a fixed snapshot of the past, these benchmarks suffer from a dual vulnerability: data contamination, stemming from the memorization of open-source training data, and performance saturation, as rapidly evolving LLMs quickly master fixed difficulty levels. Consequently, the field urgently requires an automated, scalable pipeline capable of continuously transforming code contexts into diverse tasks. Such a framework must support dynamic difficulty scaling to prevent saturation and employ novel transformations to mitigate memorization, ensuring evaluation remains a rigorous, moving target aligned with the frontiers of code intelligence.

To address these challenges, we introduce CORECODEBENCH, a **Co**nfigurable **Re**pository-level Benchmark designed to dissect LLM coding capabilities through atomized tasks. Leveraging the novel COREPIPE framework, we automatically extract and transform 12 diverse Python repositories into a comprehensive suite of 1,524 evaluation tasks. Our approach offers three key advantages: (1) **Multi-dimensional Capability Isolation**: By generating six distinct task types on the same code context, we explicitly quantify performance disparities across distinct cognitive demands, distinguishing between generation, reasoning, and planning. (2) **Controllable Difficulty**: We dynamically modulate task complexity by manipulating factors such as mask length and dependency depth, thereby mitigating performance saturation and ensuring sustainable evaluation. (3) **Superior Data Quality**: Our automated pipeline COREPIPE ensures high reliability without manual intervention. CoreCodeBench achieves a 78.55% validity yield, significantly surpassing the 31.7% retention rate of manually filtered benchmarks like SWE-bench-Verified (OpenAI, 2024).

To validate our framework, we conduct extensive experiments on a wide range of state-of-the-art (SoTA) LLMs. Our analysis probes the underlying nature of code intelligence, revealing a significant performance misalignment across distinct cognitive demands. Crucially, the distinct performance rankings observed here compared to other benchmarks suggest that our taxonomy captures distinct dimensions of capability overlooked by monolithic evaluations. Furthermore, we demonstrate the benchmark's sustainable scalability through graded difficulty levels, verifying that our complexity adjustments effectively mitigate performance saturation and maintain rigorous challenges for evolving models. Finally, the reliability of our automated pipeline is corroborated by fine-tuning experiments, where the successful acquisition of capabilities by smaller models confirms that CoreCodeBench generates canonical and valid supervision. In summary, our key contributions are as follows:

- We propose a fine-grained taxonomy that disentangles programming proficiency into different tasks with distinct cognitive demands, moving beyond monolithic metrics to diagnose specific cognitive bottlenecks in LLMs.
- We introduce CoreCodeBench, a contamination-resilient benchmark constructed via COREPIPE. Beyond achieving a 78.55% validity yield, it ensures sustainable evaluation by dynamically scaling difficulty to mitigate performance saturation.
- We uncover a critical capability misalignment by systematically analyzing the inter-dependencies between distinct cognitive demands, thereby elucidating the internal structure of code intelligence and mapping how these dimensions correlate and diverge.

2 COREPIPE Framework

To construct CoreCodeBench, we develop COREPIPE, an automated framework designed to transform static repositories into dynamic, atomized evaluation tasks. As illustrated in Figure 2, the framework proceeds through three stages: context extraction, atomic task generation with cognitive isolation, and composite task scaling. This pipeline guarantees rigorous verifiability while enabling dynamic difficulty scaling.

2.1 Repository Context Extraction

We select 12 open-source Python repositories from PyPI (detailed in Appendix A) based on three rigorous criteria: (1) *Activeness*: maintained within the

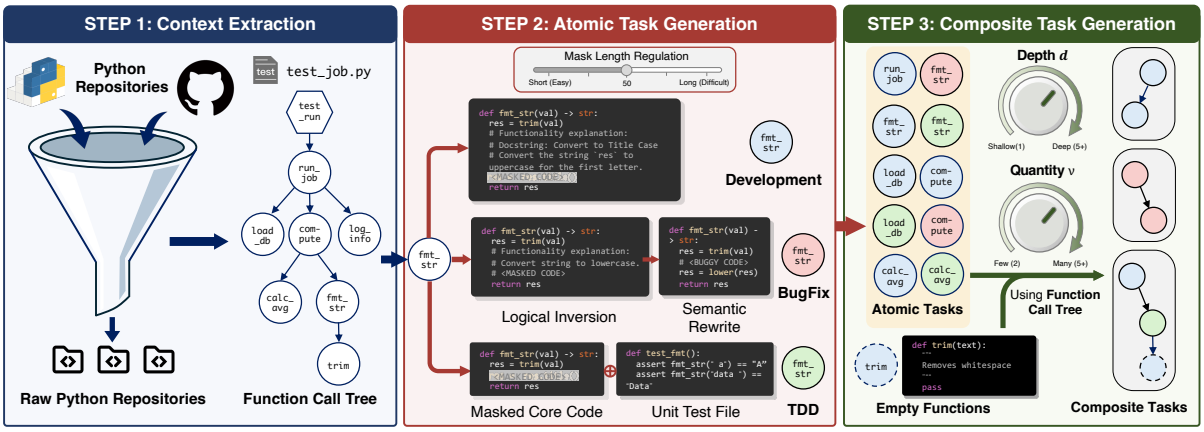


Figure 2: **Overview of the CorePipe Framework.** (i) *Context Extraction* builds verifiable Function Call Trees from unit tests. (ii) *Atomic Task Generation* isolates cognitive demands (Dev., BugFix, TDD) within identical contexts. (iii) *Composite Task Scaling* aggregates atomic tasks into subgraphs, modulating difficulty via Dependency Depth (d) and Task Quantity (ν) to prevent saturation.

last six months; (2) *Test Coverage*: test files constitute $> 15\%$ of the codebase; and (3) *Complexity*: $> 5,000$ LOC with cross-module dependencies.

To operationalize these repositories, we establish a precise mapping between source implementation and unit tests. We employ a hybrid approach combining LLM-based structural analysis with automated rule-based matching to generate `<source, test>` pairs. Subsequently, we perform dynamic tracing using a customized `pycallgraph` (Kaszuba, 2016) to construct a **Function Call Tree**. This tree structure, where nodes represent functions and edges represent invocation dependencies, serves as the backbone for both identifying core logic and composing complex tasks.

2.2 Atomic Task Generation

We generate atomic tasks by isolating single-function logic into distinct engineering scenarios. To ensure resilience against data contamination, our pipeline applies fine-grained transformations that alter the code's surface form and structural context. This prevents LLMs from simply recalling exact solutions from pre-training data, compelling them to reason grounded in the modified context.

Core Code Identification. To ensure evaluation substance, we filter out trivial utilities and focus on **core functions**. We prompt LLMs to identify semantically central AST blocks, filtering out trivial utilities. Crucially, we modulate the difficulty of the resulting tasks by varying the mask length of these blocks. We then validate the centrality of these blocks by executing unit tests on the masked code; only functions where masking triggers test failures are retained, ensuring the missing logic is essential for correctness. Manual inspection of 50 random samples confirms 100% accuracy in capturing essential logic, validating the reliability of this automated approach.

Development. We mask the identified core code blocks and employ GPT-4o (OpenAI, 2024) to generate structured functional descriptions. Crucially, our ablation study confirms that model rankings are consistent across four distinct generator backbones (Spearman's $\rho = 1.0$, see Appendix B). This validates that our metrics reflect intrinsic model capabilities, independent of generator-specific stylistic biases. To ensure quality, we introduce Claude-3.5-Sonnet (Anthropic, 2024b) as a critic model to provide feedback. The generator refines the description based on this feedback. Finally, we apply an Information Gain (IG) filter (detailed in Appendix C) to discard samples where the generated explanation fails to provide effective guidance compared to the code context alone.

BugFix. To evaluate the capability to diagnose and rectify logic errors, we extend the Development tasks by replacing the correct implementation with a buggy version. Unlike simple syntactic perturbations (Liu et al., 2025b) or adversarial injections targeting model blind spots (Ibrahimzada et al., 2025), which often lack semantic coherence or human-like intent, we employ a cascaded logic-implementation synthesis. Specifically, we prompt advanced models (e.g., GPT-4o) to design complex logical fallacies (e.g., boundary neglect, state mismanagement), and subsequently leverage smaller models (e.g., Qwen3-Coder30B) to implement these flawed

designs. This two-stage approach leverages the reasoning depth of large models to ensure logical complexity, while utilizing the stochasticity of smaller models to introduce natural implementation noise. Consequently, this bypasses the bias of LLMs to generate bug, yielding bugs that manifest as realistic developer slips rather than artificial artifacts (more details and case studies in Appendix D).

Test-Driven Development (TDD). TDD task evaluate the ability to implement logic strictly based on verification constraints (Mathews and Nagappan, 2024; Ahmed et al., 2024). We directly utilize the core code blocks identified previously, masking the implementation while providing the corresponding unit tests as the specification. This paradigm is particularly relevant to modern LLM-assisted programming workflows, where models must align generated code with rigid pre-defined tests to ensure reliability.

2.3 Composite Task Generation

To bridge the gap between isolated atomic tasks and real-world engineering, we introduce Composite Tasks (instantiated as *multi-function problems*), which require models to reason over a subgraph of the function call tree rather than isolated nodes.

We define a composite task as a subgraph of the call tree containing n atomic problems. To simulate realistic scenarios where primary logic relies on supporting components, we introduce **Empty-Function** nodes specifically for Dev. and TDD tasks. These nodes are integral parts of the subgraph but are stripped to their function signatures, representing auxiliary utilities that must be implemented synchronously to support the main objectives. Crucially, to prevent under-specification, we verify that the execution trace of the provided tests traverses these nodes, ensuring the main task strictly depends on their implementation. Furthermore, we enforce strict connectivity constraints during subgraph sampling to ensure that all selected nodes form a valid, executable dependency chain (detailed generation algorithms in Appendix E).

Crucially, the complexity of composite tasks is explicitly controlled via two hyper-parameters to mitigate performance saturation: (1) **Dependency Depth** (d), which regulates the scope of context by limiting the maximum depth of the call chain; and (2) **Task Quantity** (ν), which defines the total number of functions ($2 \leq n \leq \nu$) requiring simultaneous modification. By systematically adjusting d and ν, we create a graded difficulty scale that ensures the benchmark remains challenging for rapidly evolving models.

Table 1: **Statistics of CoreCodeBench**. # Func and # Lines denote the average number of functions and gold solution lines, respectively.

Category	Task Type	# Func	# Lines	# Prob
Atomic	Development	1.00	17.00	511
	BugFix	1.00	38.00	315
	TDD	1.00	14.00	278
Composite	Multi-Dev	3.85	53.92	167
	Multi-BugFix	2.00	62.34	10
	Multi-TDD	4.07	67.30	152
	Difficult	4.75	65.66	91

3 CoreCodeBench Dataset

Benchmark Composition. CoreCodeBench comprises 1,524 problems from 12 Python repositories. As shown in Table 1, tasks are categorized into Atomic (single-function) and Composite (orchestrating interdependent functions via call-tree subgraphs) levels across Development, BugFix, and TDD types. Notably, we retain the *Multi-BugFix* task despite its limited size (10) to preserve taxonomic symmetry, acknowledging the high difficulty of synthesizing valid interdependent bugs. Additionally, we introduce a *Difficult* subset (task quantity $\nu = \infty$) to probe the reasoning ceiling of current models. Dataset details, prompts, and robustness checks are provided in Appendix F, G, and H.

Evaluation Protocol. We evaluate performance using unit tests. Crucially, we perform a pre-generation execution (denoted as *retest*) on the initial masked or buggy code. This step records N_{retest}, the number of tests that pass *without* any model modification. We employ two metrics: (1) **AC@1** (Chen et al., 2021): The percentage of problems where the generated solution passes all unit tests. (2) **AC Rate**: A fine-grained metric measuring the proportion of fixed test cases:

$$\text{AC Rate} = \frac{N_{\text{pass}} - N_{\text{retest}}}{N_{\text{total}} - N_{\text{retest}}}, \quad (1)$$

where N_{pass} is the number of passed tests after generation. By subtracting N_{retest}, this metric strictly measures the model's contribution to fixing previously failing tests.

4 Experiments

We conduct a comprehensive evaluation to validate the quality of CoreCodeBench and probe the code intelligence of SoTA LLMs. Our analysis addresses four research questions:

- **RQ1**: Does the automated CorePipe generate high-quality, reliable evaluation tasks?
- **RQ2**: Do LLMs exhibit consistent proficiency across distinct cognitive demands?
- **RQ3**: Can CoreCodeBench effectively mitigate performance saturation and sustain challenge through dynamic complexity modulation?
- **RQ4**: Does our fine-grained taxonomy capture dimensions of capability overlooked by existing monolithic benchmarks?

4.1 Experimental Setups

We evaluate a comprehensive suite of 9 SoTA LLMs. Our selection includes leading proprietary models, including GPT-5.2 (OpenAI, 2025), Claude-4.5-Opus (Anthropic, 2025b), Gemini-3-Pro (DeepMind, 2025), and Doubao-Seed1.6 (Seed, 2025), as well as high-performance open-weights models, including Qwen3-Coder-480B-A3B-Instruct (Qwen, 2025a), GLM-4.6 (Z.ai, 2025), Kimi-K2 (Kimi et al., 2025), MiniMax-M2 (MINIMAX, 2025) and DeepSeek V3.2 (DeepSeek-AI et al., 2025). For brevity, we hereafter refer to these models using only their family names and versions. Notably, for the correlation analyses in Section 4.6, we extend this set with 3 additional LLMs to enhance statistical reliability. All evaluations are performed in a zero-shot setting using greedy decoding, detailed implementation settings is in Appendix I.

4.2 Reliability of CorePipe (RQ1)

Human Inspection. We conduct a large-scale manual inspection on 360 development problems, covering $360/511 = \mathbf{70.5}\%$ of the single-dev tasks. Experienced engineers evaluate the problems based on readability, accuracy, and completeness (criteria detailed in Appendix J). This inspection yields a **78.55%** qualification rate (with inter-annotator agreement rate > 95%), significantly surpassing the 31.7% retention rate of manually curated benchmarks like SWE-bench-Verified (OpenAI, 2024). This result confirms that CorePipe's automated mechanisms produces high-fidelity tasks suitable for reliable evaluation. For users requiring maximum precision, we also release the manually verified subset as `CoreCodeBench-Dev-Verified`. Note that TDD, BugFix and multi-function tasks are derived deterministically from existing code and tests without LLM-generated context, ensuring inherent validity without the need for manual text verification.

Table 2: **Leaderboard on Atomic Tasks.** Rate indicates AC Rate (%). Best in **bold**, second underlined.

Type	Model	Dev.		BugFix		TDD	
		AC@1	Rate	AC@1	Rate	AC@1	Rate
API	Gemini-3-Pro	67.24	**87.72**	**62.77**	**78.58**	**73.39**	**89.33**
	Claude-4.5	**68.26**	86.56	51.66	63.25	61.24	83.92
	GPT-5.2	57.66	81.80	53.56	70.92	67.35	83.27
	Seed-1.6	39.70	69.35	41.57	64.70	37.93	52.90
Open-Source	Kimi-K2	58.95	86.34	29.57	45.63	50.99	79.45
	DeepSeek-V3.2	62.06	87.16	41.00	60.83	57.59	83.04
	Qwen3-Coder	57.35	83.96	36.46	55.03	54.19	83.96
	GLM-4.6	52.56	83.26	42.46	63.68	65.23	87.33
	MiniMax-M2	23.12	53.50	18.56	30.84	33.17	61.96

Table 3: **Leaderboard on Composite Tasks.** Rate indicates AC Rate (%). Best in **bold**, second underlined.

Type	Model	Dev.		BugFix		TDD	
		AC@1	Rate	AC@1	Rate	AC@1	Rate
API	Gemini-3-Pro	**18.44**	**35.08**	0.00	15.99	**20.67**	**42.16**
	Claude-4.5	16.06	27.09	0.00	13.85	17.80	25.51
	GPT-5.2	3.82	9.34	0.00	13.85	13.76	20.34
	Seed-1.6	0.30	7.34	0.00	13.85	7.61	20.87
Open-Source	Kimi-K2	8.78	25.06	0.00	2.14	5.63	16.21
	DeepSeek-V3.2	15.36	32.48	0.00	13.85	9.98	27.80
	Qwen3-Coder	12.80	25.89	0.00	**18.66**	8.77	21.01
	GLM-4.6	6.76	24.77	0.00	15.99	15.91	31.32
	MiniMax-M2	0.77	5.30	0.00	0.00	2.49	10.99

Fine-tuning Validation. To further corroborate that CoreCodeBench provides canonical supervision, we perform a controlled fine-tuning experiment. We adopt a repository-level split (11 for training, 1 for testing) to prevent data leakage. We fine-tune Qwen3-8B (Qwen, 2025b) on the training split and evaluate their performance on the unseen test repositories. Training details and experiment results are available in Appendix K. The result yields significant improvements on 5 metrics across all three atomic tasks (e.g., Dev. AC@1 increases by 19.15%), confirming that CoreCodeBench serves as a canonical supervision signal.

4.3 Main Results: Overall Performance

Table 2 reports the performance on atomic tasks, while Table 3 summarizes the results on composite tasks. Confidence intervals and results for addi-

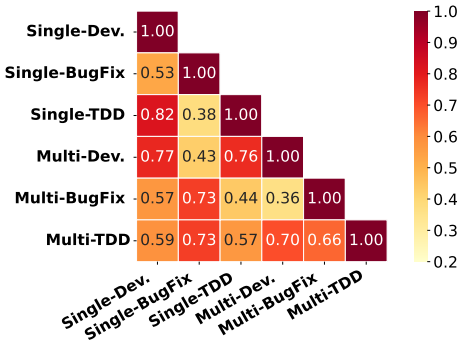


Figure 3: Pearson correlation matrix across six tasks.

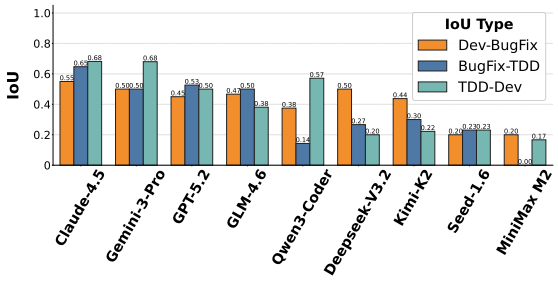


Figure 4: Quantifying Capability Misalignment via Inter-task IoU. The generally low IoU values reveal significant inconsistency between tasks.

tional LLMs are provided in Appendix L.

Atomic Tasks. As shown in Table 2, proprietary models establish a clear lead, with Gemini-3-Pro and Claude-4.5-Opus achieving top-tier performance. While open-source models like GLM-4.6 and Qwen3-Coder have narrowed the gap in generation-heavy tasks (Development and TDD), performance drops significantly on BugFix across the board. This decline is particularly pronounced for open-source models, where AC@1 scores often halve compared to Development. This suggests that diagnosing logic errors remains a distinct bottleneck compared to generation, a phenomenon we analyze further in Section 4.4.

Composite Tasks. Table 3 reveals a dramatic increase in difficulty as we move to repo-level composite tasks. Most strikingly, Multi-BugFix proves to be an extremely formidable challenge, with all models failing to solve a single instance (0.00% AC@1), though non-zero AC Rates indicate partial progress. This performance collapse correlates with a critical behavioral observation: in multi-function scenarios, models are required to complete multiple functions within a single response. Ideally, LLM would demonstrate planning by ordering implementations based on dependencies (e.g., implementing utility functions before their callers). However, all models strictly adhered to the prompt's sequence, ignoring dependency logic. This rigid linearity exposes a critical deficit in hierarchical reasoning, rendering current models ineffective in complex, interdependent environments.

4.4 Capability Misalignment (RQ2)

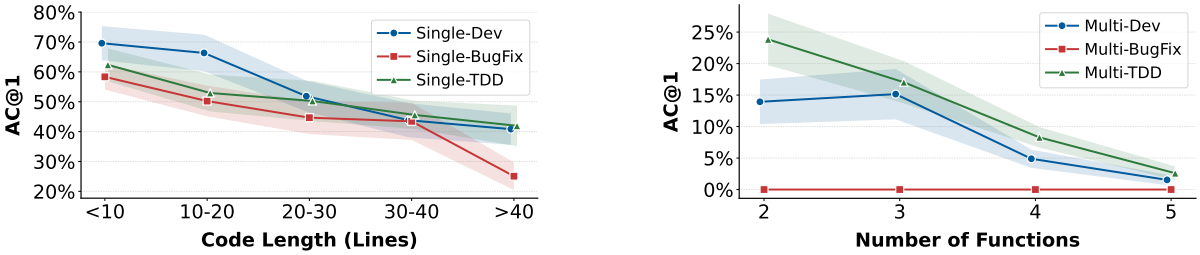
Task Correlation Analysis. To quantify the relationships between different evaluation dimensions, we analyze the performance alignment across the six task types. Specifically, we construct performance vectors for each task using the AC Rate scores of all evaluated models. We then calculate the Pearson correlation coefficient (Pearson, 1896) between every pair of task vectors, as visualized in Figure 3. Using AC Rate ensures meaningful correlation analysis even for challenging tasks like Multi-BugFix where AC@1 scores are sparse.

Inter-task Consistency. To further probe whether models possess a unified understanding of the code context, we analyze the consistency of their success across different tasks derived from the identical function using the Intersection-over-Union (IoU) metric. For any pair of tasks A and B, IoU is calculated as: $\text{IoU}(A, B) = \frac{|S_A \cap S_B|}{|S_A \cup S_B|}$, where S_A and S_B denote the sets of instances solved by the model in task A and B, respectively.

As visualized in Figure 4, the absolute IoU values reveal a pervasive lack of consistency. Even for the strongest proprietary models (e.g., Claude-4.5, Gemini-3-Pro), the overlap is less than 0.7, implying that in at least 30% of cases, proficiency is inconsistent across dimensions. The landscape for open-source models is even more fragmented and diverse. While Qwen3-Coder maintain moderate consistency between Development and TDD, their alignment with BugFix is drastically lower (< 0.2). This variance in IoU distributions confirms that different models rely on distinct, often disjoint capability profiles, lacking a robust, transferable mental model of the code.

4.5 Mitigating Performance Saturation (RQ3)

A key design goal of CoreCodeBench is to prevent performance saturation by providing controllable difficulty gradients. We analyze how model performance scales with two key complexity factors: code length and dependency scope.



(a) Impact of Code Length.

(b) Impact of Task Quantity ν.

Figure 5: **Difficulty Scaling Analysis.** (a) Performance consistently declines as the code length increases. (b) Increasing the number of interdependent functions (ν) triggers a performance collapse.

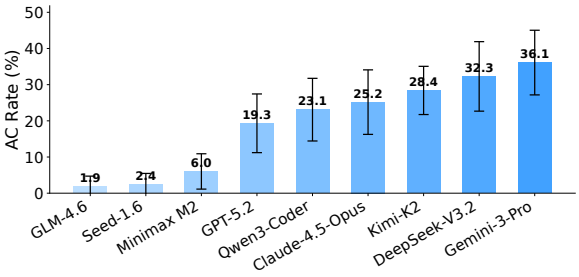


Figure 6: **Performance on CoreCodeBench-*Difficult***.

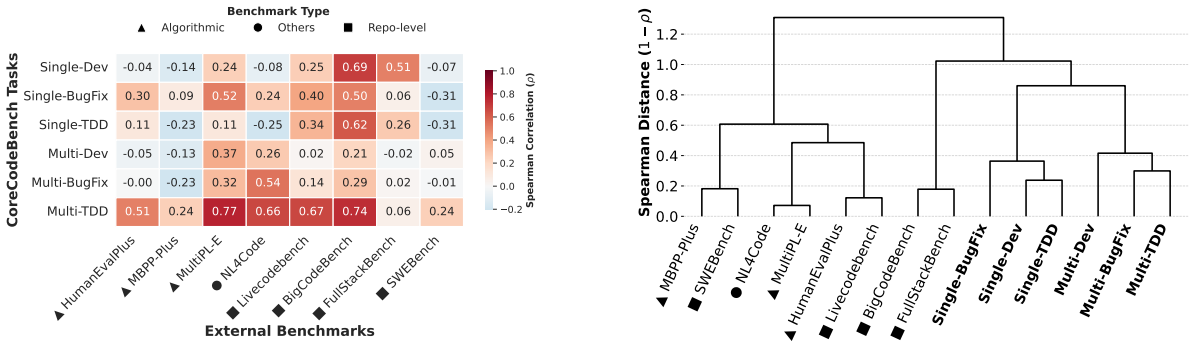
Impact of Mask Length. Figure 5(a) illustrates the relationship between the length of the masked core code block (a configurable parameter in CorePipe) and model average Pass@1 across atomic tasks. We observe a consistent negative correlation: as the mask length increases, the average AC@1 drops significantly across all tasks. Notably, BugFix (Red line) exhibits the steepest decline in long contexts (> 30 lines), falling below 25%. This suggests that while models can manage short context repairs, their reasoning capability degrades rapidly as the search space for logic errors expands. Crucially, this trend confirms that by adjusting the mask length, CorePipe can effectively modulate the difficulty of atomic tasks, tailoring the challenge to match evolving model capabilities.

Impact of Task Quantity (ν). As shown in Figure 5(b), we further investigate the effect of dependency complexity by scaling the number of interdependent functions (ν) in Multi-Function tasks. Performance degrades sharply as ν increases, revealing a clear complexity cliff. Notably, Multi-BugFix (Red line) remains near zero throughout. This is expected, as simultaneously diagnosing logic errors in just two interdependent functions ($\nu = 2$) already exceeds the reasoning capacity of current models; increasing ν further only compounds this insurmountable challenge. This confirms that manipulating ν serves as a potent lever to modulate difficulty, ensuring the benchmark remains challenging for future models. To push this to the limit, we construct the **CoreCodeBench-Difficult** subset by setting $\nu = \infty$ (unbounded task quantity). As shown in Figure 6, even the strongest models achieve AC Rates below 40% on this subset. This explicitly delineates the current ceiling of code intelligence, providing a rigorous testbed for driving future advancements.

4.6 External Validity (RQ4)

To position CoreCodeBench within the broader evaluation ecosystem, we analyze its relationship with 8 major external benchmarks, ranging from algorithmic puzzles (including HumanEvalPlus (Liu et al., 2023a), MBPP-Plus (Liu et al., 2023a) and MultiPL-E (Cassano et al., 2022)) to repository-level engineering tasks (including SWE-bench (OpenAI, 2024), BigCodeBench (Zhuo et al., 2025), LiveCodeBench (Jain et al., 2024) and FullStackBench (Cheng et al., 2024)). We compute the Spearman correlation coefficients (ρ) (Spearman, 1961) between the performance rankings of 12 LLMs (the 9 primary models plus 3 extended ones) across our six tasks and external benchmarks.

Correlation Landscape. The heatmap in Figure 7a visualizes the pairwise correlations, revealing the limited coverage of existing benchmarks. Most external datasets correlate well with only specific subsets of our tasks, failing to capture the full spectrum of engineering capabilities. For instance, algorithmic benchmarks align closely with our TDD tasks, because both paradigms rely on test-guided generation where models optimize code to satisfy rigid input-output constraints. In contrast, repository-level benchmarks exhibit a more mixed correlation pattern. This indicates that while existing benchmarks serve as specialized probes for monolithic proficiency, CoreCodeBench provides a more granular diagnosis, encompassing dimen-



(a) Correlation Heatmap.

(b) Hierarchical Clustering.

Figure 7: **Comparison with Existing Benchmarks.** (a) Heatmap shows limited alignment with external benchmarks. (b) Clustering reveals that CoreCodeBench forms a unique cluster distinct from current benchmarks.

sions that are often conflated or overlooked.

Structural Uniqueness. The hierarchical clustering dendrogram in Figure 7b further clarifies this distinction by revealing the latent taxonomy of the current benchmark landscape. In this visualization, the vertical height of the branches represents the dissimilarity between tasks; tasks that merge at lower heights are more strongly correlated. The analysis uncovers a fundamental bifurcation into two primary clusters. The traditional cluster (left) aggregates the majority of existing benchmarks, and CoreCodeBench forms a distinct branch (right). This structural separation serves as strong empirical evidence that CoreCodeBench introduces a novel evaluation perspective. Rather than merely replicating existing difficulty levels, it captures specific dimensions of engineering capability that differ significantly from the established evaluation ecosystem.

5 Related Work

Coding Capability of LLMs. LLMs have achieved remarkable proficiency in coding capabilities. On HumanEval (Chen et al., 2021), leading closed-source models like Claude-3.5-Sonnet (Anthropic, 2024a) and GPT-4o (OpenAI, 2024), as well as open-source DeepSeek-Coder-V2 (DeepSeek-AI, 2024) and Qwen2.5-Coder (Hui et al., 2024), all exceed 88% AC@1. Similar mastery is observed on MBPP (Austin et al., 2021) with scores surpassing 85%. Beyond function-level synthesis, the field has transitioned to complex software engineering; frontier models such as Claude 4.5 Opus (Anthropic, 2025b), Gemini 3 Pro (DeepMind, 2025) reportedly achieve over 70% resolution rates on SWE-bench (OpenAI, 2024).

Repository-level Code Benchmarks. Existing repository-level benchmarks are typically designed for isolated scenarios, such as code completion (Wu et al., 2025; Niu et al., 2023), functional code generation (Hai et al., 2025; Yang et al., 2024; Fu et al., 2024), multi-stage software development (Li et al., 2024a), or specialized engineering tasks like code translation (Wang et al., 2025) and version migration (Liu et al., 2025a). Their underlying data is often derived from random masking (Liu et al., 2023b), automated extraction of GitHub issues and pull requests (Jimenez et al., 2024; Pan et al., 2024; Li et al., 2025), or manual curation (Zhuo et al., 2025; Li et al., 2024b), which renders the evaluation static, difficult to scale, and vulnerable to data contamination. Consequently, the field requires a dynamic framework to dissect the monolithic proficiency masked by static evaluations.

6 Conclusion

We introduce CoreCodeBench, a repository-level benchmark decomposing code intelligence into distinct cognitive dimensions. Leveraging the automated CorePipe, we transform repositories into 1,524 tasks with controllable difficulty and high quality without reliance on manual curation. Our extensive evaluation reveals a significant capability misalignment: models exhibit uneven proficiency across development, debugging, and planning tasks, even within the identical code context. This finding challenges the prevailing monolithic view of coding proficiency, demonstrating that robust software engineering requires the synergy of diverse, often imbalanced capabilities. Furthermore, our structural analysis confirms that CoreCodeBench captures evaluation dimensions that differ from existing algorithmic and repair benchmarks.

Limitations

Despite the automated generation capabilities of CorePipe, our framework currently relies on the presence of high-coverage unit tests within source repositories. Consequently, repositories with sparse tests cannot be processed, potentially biasing the benchmark towards well-maintained projects. Future work will explore automated test generation to broaden this scope. Additionally, due to the stringent constraints required to synthesize valid interdependent logic errors, the *Multi-BugFix* subset contains a limited number of instances (10). While valuable as a case study for extreme difficulty, its statistical power for ranking is inherently lower than other categories. Finally, CoreCodeBench focuses exclusively on Python; extending support to other languages (e.g., Java, C++) remains a critical direction to evaluate cross-lingual engineering proficiency.

References

Toufique Ahmed, Martin Hirzel, Rangeet Pan, Avraham Shinnar, and Saurabh Sinha. 2024. Tdd-bench verified: Can llms generate tests for issues before they get resolved? *Preprint*, arXiv:2412.02883.

Anthropic. 2024a. The claude 3 model family: Opus, sonnet, haiku. Accessed: 2024-05-10.

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

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

Anthropic. 2025b. Introducing claude opus 4.5. https://www.anthropic.com/news/claude-opus-4-5. Accessed: 2025-11-25.

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

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2022. Multipl-e: A scalable and extensible approach to benchmarking neural code generation. *Preprint*, arXiv:2208.08227.

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, and 39 others. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Yao Cheng, Jianfeng Chen, Jie Chen, Li Chen, Liyu Chen, Wentao Chen, Zhengyu Chen, Shijie Geng, Aoyan Li, Bo Li, and 1 others. 2024. Fullstack bench: Evaluating llms as full stack coders. *arXiv preprint arXiv:2412.00535*.

Google DeepMind. 2025. Gemini 3 pro: Best for complex tasks and bringing creative concepts to life. https://deepmind.google/models/gemini/pro/. Accessed: 2025-12-5.

DeepSeek-AI. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *Preprint*, arXiv:2406.11931.

DeepSeek-AI, Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenhao Xu, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, and 245 others. 2025. Deepseek-v3.2: Pushing the frontier of open large language models. *Preprint*, arXiv:2512.02556.

Brittany Terese Fasy, Fabrizio Lecci, Alessandro Rinaldo, Larry Wasserman, Sivaraman Balakrishnan, and Aarti Singh. 2014. Confidence sets for persistence diagrams. *The Annals of Statistics*, 42(6).

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. 2024. Codeapex: A bilingual programming evaluation benchmark for large language models. *Preprint*, arXiv:2309.01940.

Nam Le Hai, Dung Manh Nguyen, and Nghi D. Q. Bui. 2025. On the impacts of contexts on repository-level code generation. *Preprint*, arXiv:2406.11927.

Dong Huang, Jie M. Zhang, Mark Harman, Qianru Zhang, Mingzhe Du, and See-Kiong Ng. 2025. Benchmarking llms for unit test generation from real-world functions. *Preprint*, arXiv:2508.00408.

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, and 5 others. 2024. Qwen2.5-coder technical report. *Preprint*, arXiv:2409.12186.

Ali Reza Ibrahimzada, Yang Chen, Ryan Rong, and Reyhaneh Jabbarvand. 2025. Challenging bug prediction and repair models with synthetic bugs. *Preprint*, arXiv:2310.02407.

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

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? *Preprint*, arXiv:2310.06770.

Gerald Kaszuba. 2016. Python call graph. *Internet: https://pycallgraph. readthedocs. io/en/master*.

Kimi, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, Zhuofu Chen, Jialei Cui, Hao Ding, Mengnan Dong, Angang Du, Chenzhuang Du, Dikang Du, Yulun Du, Yu Fan, and 150 others. 2025. Kimi k2: Open agentic intelligence. *Preprint*, arXiv:2507.20534.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. *Preprint*, arXiv:2309.06180.

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. 2024a. Prompting large language models to tackle the full software development lifecycle: A case study. *Preprint*, arXiv:2403.08604.

Jia Li, Ge Li, Xuanming Zhang, Yihong Dong, and Zhi Jin. 2024b. Evocodebench: An evolving code generation benchmark aligned with real-world code repositories. *Preprint*, arXiv:2404.00599.

Wei Li, Xin Zhang, Zhongxin Guo, Shaoguang Mao, Wen Luo, Guangyue Peng, Yangyu Huang, Houfeng Wang, and Scarlett Li. 2025. Fea-bench: A benchmark for evaluating repository-level code generation for feature implementation. *Preprint*, arXiv:2503.06680.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023a. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36:21558–21572.

Linbo Liu, Xinle Liu, Qiang Zhou, Lin Chen, Yihan Liu, Hoan Nguyen, Behrooz Omidvar-Tehrani, Xi Shen, Jun Huan, Omer Tripp, and Anoop Deoras. 2025a. Migrationbench: Repository-level code migration benchmark from java 8. *Preprint*, arXiv:2505.09569.

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. 2025b. Mdeval: Massively multilingual code debugging. *Preprint*, arXiv:2411.02310.

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

Noble Saji Mathews and Meiyappan Nagappan. 2024. 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.

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

MINIMAX. 2025. Minimax m2 & agent: Ingenious in simplicity. https://www.minimax.io/news/minimax-m2. Accessed: 2025-10-27.

Changan Niu, Chuanyi Li, Vincent Ng, and Bin Luo. 2023. Crosscodebench: Benchmarking cross-task generalization of source code models. *Preprint*, arXiv:2302.04030.

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

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

OpenAI. 2025. Introducing gpt5.2. https://openai.com/index/introducing-gpt-5-2/. Accessed: 2025-12-11.

Zhenyu Pan, Rongyu Cao, Yongchang Cao, Yingwei Ma, Binhua Li, Fei Huang, Han Liu, and Yongbin Li. 2024. Codev-bench: How do llms understand developer-centric code completion? *Preprint*, arXiv:2410.01353.

K. Pearson. 1896. Vii. mathematical contributions to the theory of evolution.-iii. regression, heredity, and panmixia. *Philosophical Transactions of the Royal Society A*, 187:253–318.

Qwen. 2025a. Qwen3-coder: Agentic coding in the world. https://qwenlm.github.io/blog/qwen3-coder/. Accessed: 2025-07-22.

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

Seed. 2025. Seed1.6 tech introduction. https://seed.bytedance.com/en/seed1_6. Accessed: 2025-06-25.

Charles Spearman. 1961. The proof and measurement of association between two things.

Alexei Stepanov. 2015. On the kendall correlation coefficient. *Preprint*, arXiv:1507.01427.

Yanli Wang, Yanlin Wang, Suiquan Wang, Daya Guo, Jiachi Chen, John Grundy, Xilin Liu, Yuchi Ma, Mingzhi Mao, Hongyu Zhang, and Zibin Zheng. 2025. Repotransbench: A real-world multilingual benchmark for repository-level code translation. *Preprint*, arXiv:2412.17744.

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. 2025. Repomastereval: Evaluating code completion via real-world repositories. *Preprint*, arXiv:2408.03519.

Jingxuan Xu, Ken Deng, Weihao Li, and Songwei etc Yu. 2025. Swe-compass: Towards unified evaluation of agentic coding abilities for large language models. *arXiv preprint arXiv:2511.05459*.

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. 2024. Execrepobench: Multi-level executable code completion evaluation. *Preprint*, arXiv:2412.11990.

Z.ai. 2025. Glm-4.6: Advanced agentic, reasoning and coding capabilities. https://z.ai/blog/glm-4.6. Accessed: 2025-09-30.

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, and 14 others. 2025. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *Preprint*, arXiv:2406.15877.

A Source Repository Details

Table 4 summarizes the metadata and statistical characteristics of the 12 selected repositories. To ensure the benchmark's representativeness and quality, we curated these projects from PyPI[1] based on strict inclusion criteria. As shown in the table, the selected repositories exhibit significant diversity in domain, test framework (e.g., `unittest` vs `pytest`), and project structure. Despite this heterogeneity, CorePipe successfully parses and processes all repositories, demonstrating robust adaptability to varied engineering conventions.

B Robustness of Benchmarking Results to Generator Selection

We utilize GPT-4o as the primary backbone to generate structured functional descriptions. To verify that our evaluation results are not biased by this choice, we conduct a comprehensive sensitivity analysis using four distinct generator backbones: GPT-4o, Claude-3.5-Sonnet, Qwen-Plus-Latest, and Doubao-Pro-4k. As visualized in Figure 8, different backbone models exhibit distinct stylistic characteristics: GPT-4o tends to produce richer, more granular details (e.g., explicit loop conditions), whereas Claude 3.5 favors conciseness.

Despite this variation in input prompt density, our quantitative analysis demonstrates that the benchmark's discriminative power remains invariant. As illustrated in Figure 9, we evaluate four representative test models across tasks generated by each backbone. While absolute pass rates fluctuate slightly due to varying description styles (e.g., Claude-3.5 tends to be more concise), the relative ranking of the test models remains strictly invariant. Specifically, regardless of which model generated the task, the performance order is consistently: Claude-3.5 $>$ GPT-4o $>$ Doubao-Pro $>$ Qwen-Plus-Latest. This yields a perfect Spearman Rank Correlation Coefficient of $\rho = 1.0$ across all generator configurations. This empirical evidence strongly confirms that CoreCodeBench provides a stable, generator-agnostic assessment of coding proficiency.

Furthermore, we evaluate the performance of different models on development-type tasks using descriptions generated by different backbones. As illustrated in Figure 9 the absolute scores of the models fluctuate due to differences in the description styles. However, the relative ranking of the

[1] https://pypi.org/

Table 4: Repository Information.

Repo	Created Time	Latest Version	Latest Release Time	Github Link	Total Code Lines	Python Files	Test Files	Test Coverage (%)
transformers	2019/9/26	4.51.3	2025/4/14	/huggingface/transformers	971,687	1,756	712	40.55
langchain	2022/10/25	0.3.25	2025/5/3	/langchain-ai/langchain	68,790	1,329	265	19.94
datachain	2024/6/27	0.16.4	2025/5/1	/iterative/datachain/tree/main	26,777	137	57	41.61
open-iris	2023/12/14	1.5.0	2025/4/22	/worldcoin/open-iris	8,072	76	64	84.21
UniRef	2023/12/26	0.6	2023/12/26	/FoundationVision/UniRef	36,127	152	50	32.89
haystack	2023/11/25	2.13.1	2025/4/24	/deepset-ai/haystack	33,905	211	150	71.09
d3rlpy	2020/7/31	2.8.1	2025/3/2	/takuseno/d3rlpy	23,984	125	45	36.00
inference	2023/8/16	0.48.3	2025/5/6	/roboflow/inference	83,164	640	118	18.44
rdt	2018/8/23	1.16.0	2025/4/11	/sdv-dev/RDT	7,265	31	16	51.61
cloudnetpy	2019/9/13	1.75.0	2025/5/2	/actris-cloudnet/cloudnetpy	23,025	116	49	42.24
skfolio	2023/12/15	0.9.0	2025/4/5	/skfolio/skfolio	29,865	113	71	62.83
finam	2023/2/3	1.0.1	2025/4/23	/finam-ufz/finam	12,592	46	30	65.22

```
# Explanation of the functionality of this code segment:
#1. **Purpose**
...
#2. **Logic**
#- Initialize the retry counter `retries` to 0.
#- While the number of retries is within the maximum allowed attempts:
#  - Attempt to parse the input string using the `self.parser.parse()` method.
#  - If parsing succeeds, return the result.
#  - If an `OutputParserException` is raised:
#    - Check whether the maximum number of retries has been reached. If so,
      raise the exception.
#    - Increment the retry counter by one.
#    - Check whether `self.legacy` and the `retry_chain.run` method exist:
#      - If they exist, call `retry_chain.run()` to retry;
#      - otherwise, attempt to call `retry_chain.invoke()`.
#        - If the parser has a `get_format_instructions method`, use it;
#        - if a `NotImplementedError` or `AttributeError` is raised, skip the
            instructions parameter.
#  - Continue parsing with the result returned from the retry.
#3. **Exceptions**
...
```

(a) Claude 3.5

```
# Explanation of the functionality of this code segment:
#1. **Purpose**
...
#2. **Logic**
#- Initialize the retry counter `retries` to 0.
#- Enter a loop with the condition `retries` <= `self.max_retries`:
#  - Call the `self.parser.parse()` method to parse the input string.
#  - If parsing succeeds, immediately return the parsed result.
#  - If a parsing error occurs, catch the `OutputParserException` exception:
#    - Check whether the current retry count `retries` has reached the maximum
      value `self.max_retries`. If so, raise the exception and terminate the
      parsing process.
#    - Otherwise, increment the retry counter: `retries += 1`.
#    - Check whether `self.legacy` is `True` and `retry_chain` has a `run` method:
#      - If so, call the `retry_chain.run()` method, passing in the parameters
        `instructions`, `completion`, and `error` to reconstruct the input
        string and retry.
#    - If the above condition is not satisfied, call `retry_chain.invoke()`method:
#      - If `self.parser` provides a `get_format_instructions()` method, use the
formatting instructions provided by this method to construct new `instructions`.
...
```

(b) GPT-4o

Figure 8: **Qualitative comparison of generated descriptions.** GPT-4o (b) captures more granular logic details (highlighted in red) compared to the more concise Claude 3.5 (a). Despite these significant stylistic divergences, our robustness analysis confirms that model rankings remain stable.

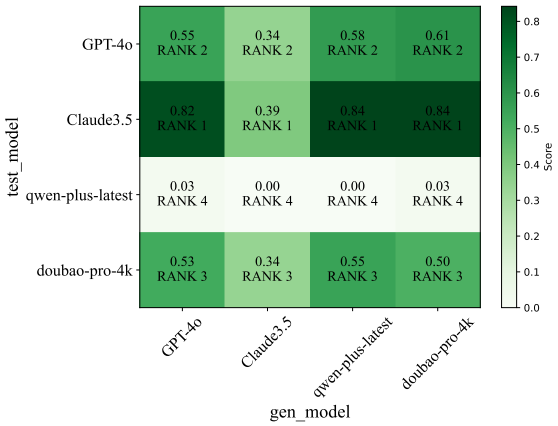


Figure 9: **Generator Sensitivity Analysis.** Performance heatmap of evaluated models across descriptions generated by different backbones. While absolute pass rates fluctuate due to prompt style, the relative ranking of models remains highly consistent, confirming the benchmark's robustness to generator choice.

models remains largely consistent and is not affected by the choice of the backbone model.

C Information Gain (IG) Filtering Protocol

To ensure that the generated natural language descriptions in development tasks provide effective guidance rather than noise, we introduce an Information Gain (IG) Score. We define the score for a given problem as:

$$\text{IG} = \text{AC Rate}_{\text{exp}} - \text{AC Rate}_{\text{no-exp}} \quad (2)$$

where AC Rate_{exp} is the average pass rate of baseline models (GPT-4o, Claude-3.5, Doubao-pro-4k, Qwen-plus-latest) when provided with the generated description, and AC $\text{Rate}_{\text{no-exp}}$ is the pass rate when provided only with the masked code context (i.e., infilling mode).

Filtering Criteria. We retain a problem if it satisfies one of the following conditions:

1. IG $>$ 0: The description positively aids the model, indicating it contains valid functional specifications.

2. AC Rate_{exp} = AC $\text{Rate}_{\text{no-exp}} = 0$: The problem is challenging for all baselines regardless of the description. We retain these as "Hard" instances, provided there is no evidence that the description is misleading (i.e., we filter out cases where AC $\text{Rate}_{\text{no-exp}} >$ AC Rate_{exp}).

After applying this filter, 48.56% of the generated candidates are retained.

Ablation Study. As shown in Table 5, we evaluate models on the discarded subset (IG ≤ 0). The results show compressed performance gaps and random fluctuations, confirming that low-IG questions fail to effectively discriminate model capabilities.

Table 5: Model Performance on Low-IG Problems.

Model	AC Rate	AC@1
Gemini-2.5-Pro	98.84	92.52
GPT-5	98.23	92.53
Doubao-Seed-1.6	85.11	71.28
qwen-plus-latest	92.90	82.11

D Qualitative Analysis of Bug Realism

To ensure the generated bugs reflect real-world development scenarios rather than artificial noise, we validate our cascaded logic-implementation synthesis approach.

Rationale for Cascaded Synthesis. Directly prompting advanced models (e.g., GPT-4o) to generate buggy code often results in *resistance to failure* due to RLHF alignment, or yields trivial syntax errors when forced. By decoupling logical fallacy design (assigned to advanced models for complex error planning, including gpt-4o, claude 3.5, etc.) from implementation (assigned to smaller models, such as QwenCoder 30B, DeepSeek 16B), we simulate a realistic scenario: a developer conceptualizing a solution but encountering implementation discrepancies. This two-stage approach leverages the reasoning depth of large models to ensure logical complexity, while utilizing the stochasticity of smaller models to introduce natural implementation noise (e.g., variable shadowing, boundary oversights), thereby maintaining the semantic structure of the code.

Bug Diversity and Characteristics. Systematic analysis reveals a diverse distribution of error types mirroring realistic developer mistakes:

- **Boundary Value Errors**: Off-by-one errors in loops or conditional checks (e.g., < vs <=).
- **Control Flow Logic**: Incorrect nesting, premature `return`, or missing termination conditions.
- **Variable Misuse**: Variable shadowing, state update errors, or incorrect variable references.
- **Exception Handling**: Omission of edge case checks (e.g., `None` types).
- **Algorithm Logic Errors**: Logic inversion or incorrect operator precedence.

This diversity significantly outperforms simple syntactic perturbation baselines in terms of realism and semantic coherence.

Case Study Comparison. We compare a bug generated by a direct GPT-4o prompt versus one by our CorePipe framework.
Original Correct Code:

```
size = get_size_dict(size)
shortest_edge = min(size["height"], size["width"
])

output_size = get_resize_output_image_size(
    image,
    size=shortest_edge,
    default_to_square=False,
    input_data_format=input_data_format
)

resized_image = resize(
    image,
    size=output_size,
    resample=resample,
    data_format=data_format,
    input_data_format=input_data_format,
    **kwargs,
)
```

Baseline (GPT-4o Direct Generation): The model introduces a trivial operator inversion (`min` → `max`). This represents a superficial error lacking semantic depth.

```
# Artificial Error: Simple Operator Flip
shortest_edge = max(size["height"], size["width"
])
```

CorePipe Generation (Ours): The cascaded approach synthesizes a compound error involving API hallucination (`size["shortest_edge"]`), parameter misalignment (`is_square`), and variable shadowing. These errors reflect plausible semantic misunderstandings typical of complex implementation tasks.

```
# Realistic Developer Slips: API Misuse, Key
Error, and Variable Shadowing
resized_image = get_resize_output_image_size(
image, size["shortest_edge"], is_square=False,
input_data_format=input_data_format)

# Superfluous Logic
if resized_image[0] > resized_image[1]:
    resized_image = resized_image[::-1]

resized_image = resize(image, size=resized_image,
 resample=resample, data_format=data_format,
input_data_format=input_data_format)
```

Finally, consistent with our validity protocols, all generated buggy codes undergo strict unit test regression validation. Only code that passes syntax checks but fails specific functional tests is retained, ensuring the bugs are executable, non-trivial, and theoretically repairable.

E Generation Rules for Composite Tasks

To construct composite tasks that simulate realistic engineering workflows, we employ a rigorous subgraph sampling and node assignment protocol. The generation process follows three key steps:

Subgraph Sampling. We first construct the global Function Call Tree $\mathcal{G} = (V, E)$ for each repository. Given the **configurable hyperparameters** for task quantity (ν) and dependency depth (d), we randomly sample a connected subgraph $\mathcal{G}' \subseteq \mathcal{G}$ such that $|V(\mathcal{G}')| = \nu$ and the depth of $\mathcal{G}'$ does not exceed d. Crucially, we verify that $\mathcal{G}'$ remains a valid call hierarchy where the root node's execution trace can reach all leaf nodes, ensuring that *no node is isolated from the testing context*.

Node Assignment Rules. Once the subgraph structure is determined, we assign specific atomic task types to each node $v \in V(\mathcal{G}')$ based on the composite task category:

- **Multi-Dev**: All nodes are assigned as *Development* tasks. To simulate dependency implementation, leaf nodes or auxiliary utilities may be converted into *Empty-Function* nodes (signature only), forcing the model to implement supporting logic from scratch.
- **Multi-BugFix**: All nodes are assigned as *BugFix* tasks. We strictly prohibit mixing Development tasks here, adhering to the real-world principle that debugging sessions typically focus on rectifying existing logic rather than implementing new features simultaneously.
- **Multi-TDD**: At least one node (typically the root or a core logic node) is assigned as a *TDD* task (implementation via test constraints). Remaining nodes can be *Development* or *Empty-Function* tasks. This hybrid composition simulates a collaborative environment where developers must align new implementations with rigid test specifications while managing dependencies.

Difficult Subset. For the CORECODEBENCH-*Difficult* subset ($\nu = +\infty$), we enforce a minimum complexity constraint of $n \geq 3$ nodes, ensuring that the task requires reasoning over a non-trivial dependency chain.

F Illustrative Examples of Atomic Tasks

Figure 10 visualizes the structure of the four atomic task types: Development, BugFix, TDD, and Empty-Function. As shown, all tasks are derived from the same function context but mask different components to isolate distinct cognitive demands:

- **Development**: Masks the function body, providing a natural language spec.
- **BugFix**: Provides a complete but logically flawed implementation.
- **TDD**: Masks the body but provides unit tests as the specification.
- **Empty-Function**: Specialized for composite tasks, this type strips the body to a bare signature, serving as a dependency node that must be implemented synchronously.

Composite tasks (multi-function problems) aggregate these atomic units into a unified prompt format, maintaining consistency in input/output interfaces.

G Prompts of Evaluation

Below, we present the evaluation prompts used for each of the six problem types.

G.1 Prompt for Atomic Tasks

Single-Development

```
Below is a code snippet containing a placeholder
 ‘<complete code here>‘. Please analyze the
provided context and description of the missing
code to generate the appropriate code block at
‘<complete code here>‘.
Please output the completed code block using
markdown format (‘‘‘python‘‘‘).
```

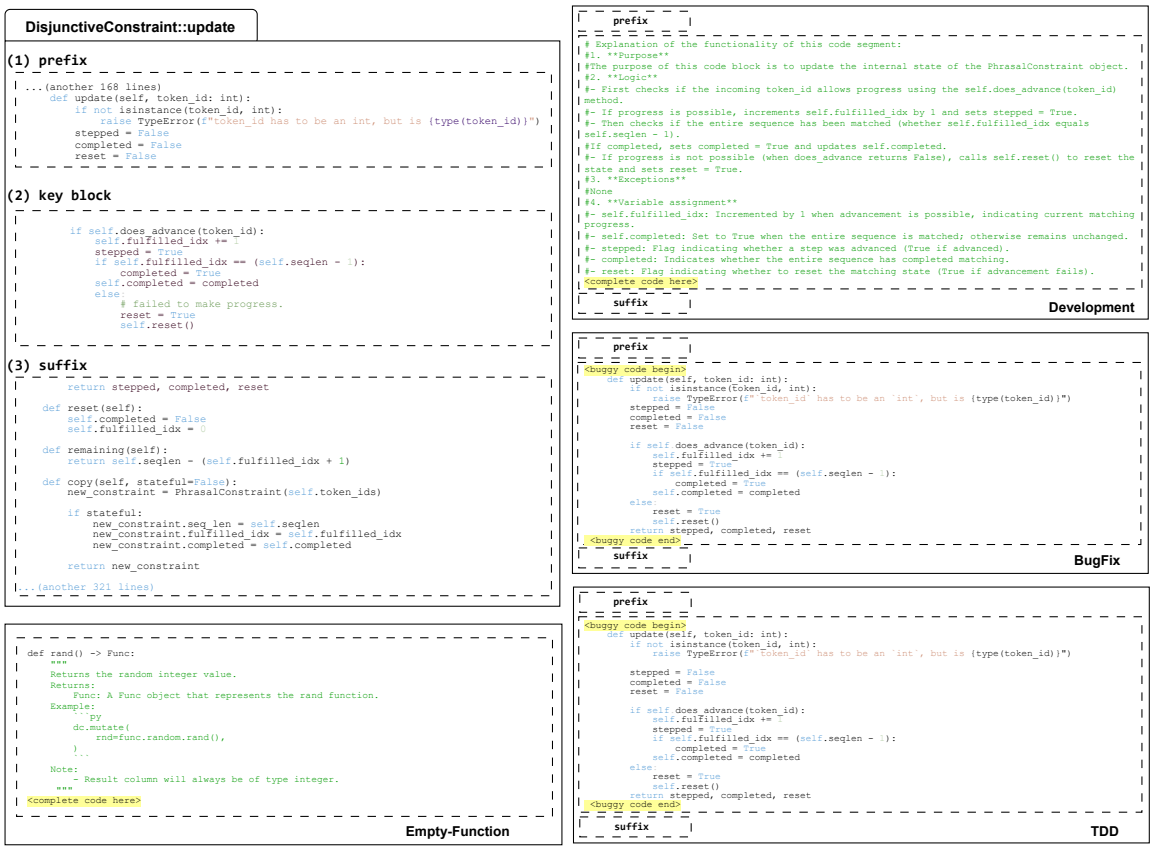


Figure 10: Illustration of atomic single-function problems.

Important: Ensure the code block you complete maintains the same indentation as the context code, meaning you need to preserve the original code's indentation. The output must exactly match the line count and structure of the input, including preserving empty lines and comment positions.
Code snippet:
```python
{prompt}
```
Please output the completed code block using markdown format (```python```). Make sure to preserve the original indentation before and after the <complete code here> placeholder. And remember don't add the signature of the function into it.

Single-BugFix

In the following code snippet, there is a buggy code section between '<buggy code begin>' and '<buggy code end>'. I've provided the corresponding unit test file and pytest error messages. Please analyze the given context and rewrite the erroneous code segment.
Please format the rewritten function block in markdown (```python```), including only the rewritten content between '<buggy code begin>' and '<buggy code end>', without including the '<buggy code begin>' and '<buggy code end>' tags.
Note: Please ensure that your completed code block maintains the indentation of the original code context.
Code snippet:
```python
{new_code}
```
Unit test code:
```python
{test_code}
```
Test error log:
```
{log}
```

Single-TDD

Below is a code file {file_name} containing a placeholder '<complete code here>'.Please analyze the provided file context and unit test information, and generate appropriate code at the '<complete code here>' location. Please output your completed code block in markdown format (```python```). The code block should only include the code at the '<completed code here>' location, without the surrounding context.

Note: Please ensure that your completed code block maintains the indentation of the surrounding code, meaning you need to preserve the original code's indentation.

```
Code file {file_name} to be completed:
‘‘‘python
{new_code}
‘‘‘
Corresponding unit test:
‘‘‘python
{test_file}
‘‘‘
```

G.2 Prompts for Composite Tasks

Output Format and Grading Logic for Composite Tasks To enable multi-function editing, we enforce a structured output format where models sequentially implement target functions wrapped in explicit ID tags (e.g., `<id> code </id>`). Our evaluation harness automatically **extracts** these tagged segments, **patches** them into the original repository context, and **executes** the test suite for verification.

Multi-Development

```
You are a code completion agent, I would provide
 you with a snippet of code, and you would need
to return the completed code segment.
the code after <ralated code> is used while
calling the code to be completed.
You need to complete code blocks after <complete
 following code> by predicting the codes after <
complete code here>, <id> label wraps the
position of the code.
Your output should include the <id></id> label,
followed by the completed code snippet enclosed
within triple backticks ‘‘‘, ensuring clarity
and proper formatting.

<related code>
<id>{id}<\id>
{related code}

<complete following code>
<id>{id}<\id>
{function code}
```

Multi-BugFix

```
In the following code snippet, the code between
<buggy code begin> and <buggy code end> contains
 bugs, <id> label wraps the position of the code.
 Please analyze the provided context and rewrite
 the faulty code segment.
The code after <related code> is used while
calling the code to be rewritten.
Your output should include the <id></id> label,
followed by the new code snippet enclosed within
 triple backticks ‘‘‘, ensuring clarity and
proper formatting.

<related code>
<id>{id}<\id>
{related code}

<complete following code>
<id>{id}<\id>
{function code}
```

Multi-TDD

```
You are a code completion agent, I would provide
 you with a snippet of code, and you would need
to return the completed code segment.
The code after <ralated code> is used while
calling the code to be completed.
You need to complete code blocks after <complete
 following code> by predicting the codes after <
complete code here>, <id> label wraps the
position of the code.
Please analyze the provided file context and the
 unit test information of the file, and generate
 an appropriate code block at the position
marked <complete code here>.
Your output should include the <id></id> label,
followed by the completed code snippet enclosed
within triple backticks ‘‘‘, ensuring clarity
and proper formatting.
Note: Please ensure that the code block you
provide as a completion matches the indentation
of the surrounding context, i.e., you need to
preserve the original code’s indentation.

<related code>
<id>{id}<\id>
{related code}

<complete following code>
<id>{id}<\id>
{function code}

The unit test information:
{test_codes}
```

H Evaluation Robustness to Prompt Variation

H.1 Evaluation Consistency under Prompt Variations

To investigate the robustness of evaluation results under prompt variation, we evaluate four mainstream LLMs using three distinct rephrased prompts for each task. Table 6 reports the Maximum Absolute Pairwise Difference (MAPD) across these variations compared to the model's 95% Confidence Interval in CoreCodeBench. For the majority of models, the performance variation (MAPD) is strictly smaller than the inherent statistical uncertainty (Confidence Interval) (Fasy et al., 2014). This suggests that CoreCodeBench provides stable assessments that are robust to surface-level prompt changes. In the case of GPT-5, slightly larger fluctuations are observed, which we attribute to the inherent non-determinism of the model's API even at temperature 0, rather than benchmark instability. Nonetheless, these variations remain within acceptable bounds for ranking purposes.

Table 6: **Model Performance Stability Under Prompt Rephrasing.** MAPD means the Maximum Absolute Pairwise Difference of evaluation results between different prompts. Conf. means 95% Confidence Interval of evaluation results.

Model	Single-Function Problem				Multi-Function Problem			
	AC Rate		AC@1		AC Rate		AC@1	
	MAPD	Conf.	MAPD	Conf.	MAPD	Conf.	MAPD	Conf.
Claude-3.7-Sonnet	0.24	2.35	0.27	2.93	3.49	4.09	0.31	3.62
GPT-5	3.97	2.05	3.99	2.83	1.16	4.54	0.61	4.32
Llama3.1-70B	0.80	2.41	0.54	2.54	0.81	3.43	0.31	2.77
Qwen3-Coder	1.88	2.24	1.18	2.95	1.72	3.41	2.44	2.65

H.2 Consistency under Context Length Variation

CoreCodeBench also exhibits strong stability with respect to variations in prompt context size. As visualized in Figure 11, both AC@1 and AC Rate exhibit highly scattered distributions across different prompt lengths, with no discernible trend. Quantitative analysis using Kendall's tau correlation coefficient (Stepanov, 2015) confirms this independence:

- Correlation between Context Size and AC@1: $\tau = 0.109$ (Negligible)
- Correlation between Context Size and AC Rate: $\tau = 0.153$ (Negligible)

These near-zero correlations indicate that the benchmark's difficulty is driven by semantic complexity rather than mere token count, ensuring fair evaluation across varying context windows. This observation validates our design choice in atomic task generation, where we explicitly modulate task difficulty via *Mask Length* rather than the total prompt length. It confirms that CoreCodeBench concerns reasoning depth, not just the ability to process long contexts.

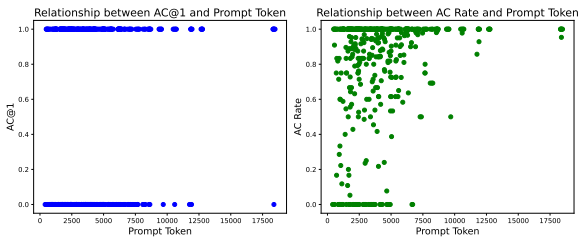


Figure 11: **Relationship between Prompt Length and Model Performance**.

I Implementation Details

I.1 Task Generation Configuration

For the identification of core code blocks, we enforce a minimum threshold of 10 lines of code to filter out trivial snippets and ensure sufficient reasoning complexity. For composite task generation, we configure the subgraph sampling parameters as follows:

- **Standard Setting**: $\nu = 6$ (task quantity), $d = 3$ (dependency depth).
- **Difficult Setting**: $\nu = \infty$ (unbounded quantity), $d = 3$.

The depth constraint ($d = 3$) is selected to balance complexity with the typical call-stack depth observed in real-world engineering.

I.2 Model Inference

To ensure reproducibility, we employ greedy decoding (temperature $= 0$, top_p $= 1.0$) for all models. For open-weights models, we utilize the `vLLM` library (Kwon et al., 2023) for efficient inference. Code blocks are extracted from model responses using standard Markdown regex patterns. To mitigate formatting artifacts, we apply heuristic post-processing, including indentation repair and function header alignment, maximizing the executability of the generated code.

I.3 Execution Environment

All evaluations are conducted within pre-configured Docker containers to guarantee isolation and reproducibility. These environments encapsulate all repository-specific dependencies and are designed to run on standard CPU instances (verified on 4 vCPU, 16GB RAM). To prevent

resource exhaustion, we enforce a strict 120-second timeout for each test execution. Dockerfiles and setup scripts will be open-sourced upon publication.

J Human Annotation

To validate the quality of CORECODEBENCH, we conduct a rigorous manual inspection on a sampled subset of tasks.

J.1 Human Annotator

We recruit three professional annotators, all holding Bachelor's degrees or higher in Computer Science or Software Engineering, with a minimum of two years of Python development experience. Annotators are compensated at competitive market rates in compliance with local labor regulations. To prevent fatigue-induced errors, the daily workload is strictly capped at 8 hours.

J.2 Annotation Workflow

The annotation process follows a strict Review-Verify-Decision protocol. We present detailed manual annotation criteria in Table 11.

J.3 Quality Control

To ensure reliability, we implement a rigorous quality control mechanism. Specifically, 50% of the annotated data undergoes cross-verification by a second annotator or a quality inspector from the author team. Disagreements are resolved through consensus meetings. This process yields an Inter-Annotator Agreement (IAA) rate exceeding 95%, demonstrating high consistency in our validation standards.

K Fine-tuning Validation Details

To empirically validate that CoreCodeBench provides high-quality, canonical supervision, we conduct a controlled fine-tuning experiment. This section details the data distribution, experimental setup, and analysis of the results.

K.1 Data Statistics and Split

We evaluate the model using a repository-level split to ensure no cross-project data leakage. Out of the 12 repositories in CoreCodeBench, we select `skfolio` as the held-out test set due to its comprehensive coverage of all three atomic task types (BugFix, Development, and TDD), while the remaining 11 repositories are used for training. Table 7 provides a summary of the task distribution across the entire benchmark used in this experiment.

Table 7: Task distribution for the fine-tuning experiment.

Split	BugFix	Development	TDD	Total
Training	297	464	222	983 (89.04%)
Test	18	47	56	121 (10.96%)
Total	**315**	**511**	**278**	**1104**

K.2 Training Methodology and Hyperparameters

The Supervised Fine-Tuning (SFT) of Qwen3-8B is conducted using a full-parameter methodology with the DeepSpeed ZeRO-3 framework. The configuration ensures the model can handle long-context repository-level information:

- Optimization: AdamW optimizer with a peak learning rate of 1.0×10^{-5} and a cosine decay scheduler.
- Hardware & Precision: Training is performed in BFloat16 precision. We use a per-device batch size of 1 with a gradient accumulation factor of 4, resulting in an effective global batch size of 32 across 8 GPUs.
- Sequence Length: The maximum sequence length is set to 32,768 tokens to accommodate complex multi-file contexts.

K.3 Fine-tuning Results and Analysis

Table 8 compares the performance of the base model and the fine-tuned version on the test set. The results demonstrate a significant performance leap. In the Development task, the model improves from 0.00% to 19.15% in AC@1, demonstrating that the model successfully acquired generalizable repository-level reasoning capabilities to navigate the unseen `skfolio` architecture. Notably, the BugFix AC Rate nearly doubles (7.78% $\rightarrow$ 15.58%), indicating that the model's generated patches become significantly more aligned with the ground truth and project constraints. These improvements on a completely unseen repository confirm that CoreCodeBench provides a high-quality, canonical supervision signal that fosters genuine generalization in software engineering tasks.

Table 8: **Model Performance before and after SFT.** We report AC@1 and AC Rate (%) for Development, Bug-Fix, and TDD.

Model	Development		BugFix		TDD	
	AC@1	Rate	AC@1	Rate	AC@1	Rate
Qwen3-8B	0.00	49.93	0.00	7.78	16.07	68.00
Qwen3-8B-FineTuned	**19.15**	**59.02**	0.00	**15.58**	**21.43**	64.51

L Confidence Intervals and Results for Additional LLMs

Tables 9 and 10 provide the detailed evaluation results and confidence intervals across various categories and problem types. Notably, we extend our reporting to include 12 models in these appendices to increase the number of sample points for the analysis in Section 4.6, thereby enhancing the statistical reliability of the ranking consistency results. The methodology used to compute these intervals is described below.

L.1 Confidence Interval Computation

For a given metric (e.g., *Rate* or *AC@1*) and task category, we obtain per-instance scores $\{x_{r,i}\}_{i=1}^{n_r}$ for each repository $r \in \{1, \ldots, R\}$. To avoid over-weighting repositories with more instances, we adopt repo-equal weighting.

Repo-equal mean. We first compute the per-repository mean

$$\bar{x}_r = \frac{1}{n_r} \sum_{i=1}^{n_r} x_{r,i},$$

and then define the overall mean as the unweighted average over repositories

$$\mu = \frac{1}{R} \sum_{r=1}^{R} \bar{x}_r.$$

Basic (CLT) 95% CI under repo-equal weighting. For each repository r, we estimate the standard error (SE) of its mean by

$$\mathrm{SE}_r = \frac{s_r}{\sqrt{n_r}}, \qquad s_r = \sqrt{\frac{1}{n_r - 1} \sum_{i=1}^{n_r} (x_{r,i} - \bar{x}_r)^2}.$$

We then combine repository-level uncertainties (treating repositories as independent) as

$$\mathrm{SE} = \sqrt{\frac{1}{R^2} \sum_{r=1}^{R} \mathrm{SE}_r^2},$$

and report the normal-approximation 95% confidence interval

$$\mu \pm z_{0.975}\,\mathrm{SE},$$

where $z_{0.975} \approx 1.96$.

M Full Prompts for CorePipe

M.1 Prompts of Repository Preprocessing

We employ Claude3.5 to analyze the file structure of each repository and automatically identify the main test directories and the source code directories. The prompt used is shown below:

```
Below is the file tree of a code repository:
{file_structure}

Please analyze the given file names and paths to
 identify the corresponding relationships
between source code and test files (paying
special attention to paths containing /test/, /
unit/, or /unittest/), and provide the output in
 JSON format. Note that the correspondence must
be based on root path relationships (for example,
 if both transformers/test/repo/ and
transformers/test/utils/ exist, select
transformers/test/). If specific unit tests
exist, the relationship should be detailed to
the unit test folder (such as unit), and the
correspondence can tolerate some missing files
as long as the files generally correspond. If
there are no similar corresponding relationships,
 please output an empty JSON object.

Example Input:
‘‘‘
- mlflow/gateway.py
- mlflow/gateway/providers.py
- mlflow/gateway/schemas.py
- mlflow/gemini.py
- mlflow/groq.py
- tests/test_gateway.py
- tests/gateway/test_providers.py
- tests/gateway/test_schemas.py
- mlflow/core/pipeline.py
- mlflow/core/pipeline/graph.py
- core_tests/pipeline.py
- core_tests/pipeline/graph.py
‘‘‘
Example Output:
‘‘‘
{
    "repo_name": "mlflow",
    "testcase_dir_mapping":{
        "mlflow/": "tests/",
        "mlflow/core/": "core_tests/",
    },
}
‘‘‘

Note that after obtaining the mapping, perform a
 check to merge paths for repeated occurrences
of upper-level directories; remove paths for non
-core code segments (such as cli, community,
_sdk, _cli/, etc.); and merge paths in cases
where possible. For example:
```

Table 9: Leaderboard on Atomic Tasks. We report AC@1 and Rate (%) with their 95% CI for Overall, Development, BugFix, and TDD, where CI denotes the 95% confidence interval half-width (margin) shown after the "±".

Type	Model	Overall		Development		BugFix		TDD	
		AC@1 ± CI	Rate ± CI	AC@1 ± CI	Rate ± CI	AC@1 ± CI	Rate ± CI	AC@1 ± CI	Rate ± CI
API	Gemini-3-Pro (DeepMind, 2025)	68.03 ± 2.75	85.36 ± 1.83	67.24 ± 4.92	87.72 ± 2.60	62.77 ± 7.61	78.58 ± 5.46	73.39 ± 6.28	89.33 ± 3.68
	Claude-4.5-Opus (Anthropic, 2025b)	61.87 ± 2.87	78.54 ± 2.22	68.26 ± 4.73	86.56 ± 2.66	51.66 ± 7.79	63.25 ± 6.81	61.24 ± 6.69	83.92 ± 3.87
	GPT-5.2 (OpenAI, 2025)	58.70 ± 2.91	78.70 ± 2.12	57.66 ± 5.16	81.80 ± 3.11	53.56 ± 7.70	70.92 ± 6.41	67.35 ± 6.60	83.27 ± 4.09
	Claude-3.7-Sonnet (Anthropic, 2025a)	43.75 ± 2.93	70.01 ± 2.35	52.81 ± 5.20	82.94 ± 3.18	27.90 ± 7.45	45.63 ± 7.38	45.11 ± 12.74	70.30 ± 11.70
	Seed-1.6 (Seed, 2025)	39.22 ± 2.88	64.69 ± 2.50	39.70 ± 5.05	69.35 ± 4.09	41.57 ± 8.02	64.70 ± 6.73	37.93 ± 6.51	52.90 ± 5.78
	Qwen3-Max-Instruct (Qwen, 2025b)	47.74 ± 2.95	70.95 ± 2.39	58.72 ± 5.14	84.30 ± 3.07	29.54 ± 7.70	41.15 ± 7.59	47.18 ± 12.86	75.80 ± 11.64
Open-Source	Kimi-K2 (Kimi et al., 2025)	47.83 ± 2.95	73.47 ± 2.27	58.95 ± 5.04	86.34 ± 2.67	29.57 ± 5.86	45.63 ± 6.49	50.99 ± 6.77	79.45 ± 4.00
	Deepseek-V3.2 (DeepSeek-AI et al., 2025)	53.89 ± 2.94	79.08 ± 2.03	62.06 ± 5.09	87.16 ± 2.58	41.00 ± 7.60	60.83 ± 6.99	57.59 ± 6.70	83.04 ± 4.01
	Qwen3-Coder-480B-A3B-Instruct (Qwen, 2025a)	50.45 ± 2.95	76.77 ± 2.12	57.35 ± 5.21	83.96 ± 3.20	36.46 ± 7.66	55.03 ± 7.05	54.19 ± 6.80	83.96 ± 3.57
	GLM-4.6 (Z.ai, 2025)	51.63 ± 2.95	78.94 ± 2.01	52.56 ± 5.24	83.26 ± 3.18	42.46 ± 7.84	63.68 ± 6.83	65.23 ± 6.49	87.33 ± 3.20
	Minimax-M2 (MINIMAX, 2025)	22.01 ± 2.45	46.40 ± 2.64	23.12 ± 4.35	53.50 ± 4.60	18.56 ± 5.66	30.84 ± 6.63	33.17 ± 5.67	61.96 ± 5.17
	Llama-3.1 (Meta, 2024)	24.64 ± 2.54	58.09 ± 2.46	33.40 ± 5.20	69.15 ± 3.82	17.37 ± 6.37	33.19 ± 6.79	32.25 ± 12.69	69.51 ± 11.57

Table 10: Leaderboard on Composite Tasks. We report AC@1 and Rate (%) with their 95% CI, where CI denotes the 95% confidence interval half-width (margin) shown after the "±".

Type	Model	Overall		Development		BugFix		TDD	
		AC@1 ± CI	Rate ± CI	AC@1 ± CI	Rate ± CI	AC@1 ± CI	Rate ± CI	AC@1 ± CI	Rate ± CI
API	Gemini-3-Pro (DeepMind, 2025)	20.73 ± 4.39	40.30 ± 4.55	18.45 ± 5.91	35.08 ± 6.69	0.00 ± 0.00	15.99 ± 16.62	20.67 ± 7.31	42.16 ± 8.03
	Claude-4.5-Opus (Anthropic, 2025b)	16.46 ± 4.02	26.83 ± 4.21	16.06 ± 6.90	27.09 ± 6.96	0.00 ± 0.00	13.85 ± 15.08	17.80 ± 5.10	25.51 ± 5.26
	GPT-5.2 (OpenAI, 2025)	9.15 ± 3.12	16.15 ± 3.56	3.82 ± 3.53	9.34 ± 4.45	0.00 ± 0.00	13.85 ± 15.08	13.76 ± 6.19	20.34 ± 6.52
	Claude-3.7-Sonnet (Anthropic, 2025a)	12.80 ± 3.62	29.83 ± 4.09	13.21 ± 5.92	29.23 ± 6.62	0.00 ± 0.00	16.99 ± 16.48	12.16 ± 5.72	28.93 ± 7.14
	Seed-1.6 (Seed, 2025)	4.88 ± 2.33	15.30 ± 3.24	0.30 ± 0.58	7.34 ± 4.05	0.00 ± 0.00	13.85 ± 15.08	7.61 ± 4.59	20.87 ± 6.70
	Qwen3-Max-Instruct (Qwen, 2025b)	10.37 ± 3.30	28.92 ± 3.97	6.01 ± 4.31	23.93 ± 5.81	0.00 ± 0.00	15.99 ± 16.62	13.83 ± 6.25	31.75 ± 7.02
Open-Source	Kimi-K2 (Kimi et al., 2025)	8.84 ± 3.08	22.88 ± 3.76	8.78 ± 4.61	25.06 ± 6.61	0.00 ± 0.00	2.14 ± 7.00	5.63 ± 3.75	16.21 ± 5.21
	Deepseek-V3.2 (DeepSeek-AI et al., 2025)	14.02 ± 3.76	31.68 ± 4.11	15.36 ± 5.93	32.48 ± 7.31	0.00 ± 0.00	13.85 ± 15.08	9.98 ± 4.90	27.80 ± 6.21
	Qwen3-Coder-480B-A3B-Instruct (Qwen, 2025a)	9.45 ± 3.17	21.30 ± 3.68	12.80 ± 6.00	25.89 ± 6.26	0.00 ± 0.00	18.66 ± 15.74	8.77 ± 4.35	21.01 ± 6.50
	GLM-4.6 (Z.ai, 2025)	11.89 ± 3.51	27.45 ± 4.02	6.76 ± 4.13	24.77 ± 5.82	0.00 ± 0.00	15.99 ± 16.62	15.91 ± 6.56	31.32 ± 6.94
	Minimax-M2 (MINIMAX, 2025)	2.13 ± 1.57	8.76 ± 2.54	0.77 ± 1.51	5.30 ± 2.94	0.00 ± 0.00	0.00 ± 0.00	2.49 ± 2.17	10.99 ± 4.33
	Llama-3.1 (Meta, 2024)	6.10 ± 2.59	18.98 ± 3.34	5.28 ± 3.67	18.67 ± 4.79	0.00 ± 0.00	17.66 ± 15.88	6.24 ± 4.19	16.21 ± 4.95

```
‘‘‘
{
    "repo_name": "langchain",
    "testcase_dir_mapping": {
        "libs/cli/langchain_cli/": "libs/cli/
tests/unit_tests/",
        "libs/community/langchain_community/": "
libs/community/tests/unit_tests/",
        "libs/core/langchain_core/": "libs/core/
tests/unit_tests/",
        "libs/langchain/langchain/": "libs/
langchain/tests/unit_tests/",
        "libs/partners/anthropic/
langchain_anthropic/": "libs/partners/anthropic/
tests/unit_tests/",
        "libs/partners/chroma/langchain_chroma
/": "libs/partners/chroma/tests/unit_tests/",
        "libs/partners/exa/langchain_exa/": "
libs/partners/exa/tests/unit_tests/",
        "src/transformers/": "tests/",
        "src/transformers/models/": "tests/
models/",
        "src/transformers/benchmark/": "tests/
benchmark/",
        "inference_sdk/": "tests/inference_sdk/
unit_tests/",

        "inference/core/": "tests/inference/
unit_tests/core/",
        "inference/enterprise/": "tests/
inference/unit_tests/enterprise/",
        "inference/models/": "tests/inference/
unit_tests/models/",
        "inference/core/workflows/": "tests/
workflows/unit_tests/"
    }
}
‘‘‘
No explanations are needed, just output in JSON
format and using ‘‘‘ ‘‘‘.
‘‘‘
{
    "repo_name": "langchain",
    "testcase_dir_mapping": {
        "libs/core/langchain_core/": "libs/core/
tests/unit_tests/",
        "libs/langchain/langchain/": "libs/
langchain/tests/unit_tests/",
        "src/transformers/": "tests/",
        "inference/": "tests/inference/
unit_tests/"
    }
}
‘‘‘
```

Our approach supports cases with multiple root directories, such as repositories such as `langchain`, which contain both source code and embedded packages (e.g., `langchain` and `langchain_core`).

After determining the main test and source direc-

tories, we traverse all files within these directories to establish fine-grained mappings between individual test files and their corresponding source files. Once valid mappings are identified, we execute the test files in the environment to verify their usability. Additionally, we record the number of test cases in each test file, which is later used to calculate the AC Rate.

We also use Claude-3.7-Sonnet to choose core code for the target function, requiring it to contain key functionality, external calls, algorithms, or core logic. Code consisting only of simple assignments or mechanical processing is excluded. Prompts for core code selection is shown below:

```
The definition of key code blocks is as follows:
- Code sections that implement the main
functionality of the function and directly
determine whether the function can achieve its
intended goal;
- Code sections whose execution efficiency
significantly impacts the function's performance.


Based on the code of function {func}, identify
the key code blocks within block {recur}, and
output the block_ids of its sub-key code blocks.
 The total number of lines in the selected code
blocks should not exceed 60 lines, so please
select carefully to ensure the most important
parts are chosen.

Output format:
If you select multiple **consecutive** blocks,
please output a list of block_ids:
‘‘‘python
blocks = ["blockid1", "blockid2", ...]
‘‘‘
If the function is relatively simple and only
contains initialization or return statements, it
 means there are no key code blocks. In this
case, please output:
‘‘‘python
blocks = None
‘‘‘
Do not include additional comments in the code
section; only output the blockid(s).

Please select key code blocks from the sub-
blocks of the {recur} code block.

Function code:
{code['func_code']}

Function block information:
{code['block_info']}
```

To validate the capability of LLMs to select core code, we randomly sample 50 generated problems for manual inspection and find that all samples (100%) meets our standards for core code selection, demonstrating the accuracy and dependability of our process.

M.2 Prompts for Single-Dev. Generation

Prompt for Explanation Generation

```
Please analyze the provided code block based on
its context, and output its functionality using
concise language in the given format (do not
include extra content):
1. **Purpose**
   Describe the main goal of the code block and
its role within the entire program. Specifically,
 what is its responsibility within the current
function?
2. **Logic**
   Elaborate on the core logic and operational
process of the code block. For all conditional
branches (if statements), explain them one by
one.
   If complex variable updates are involved, use
 Markdown format for formulas to represent these
 mathematical calculations.
   If variables from previous sections of the
code block are used, try to describe using their
 variable names, enclosing them in backticks.
Functions should be enclosed in backticks as
well, and can be in the form ‘function_name(
arguments)‘ or ‘function_name‘, without causing
ambiguity such as ‘function_name()‘ which might
lead to misunderstanding.
3. **Exceptions**
   If the code block under analysis throws
exceptions, explain its exceptional cases and
types. If no exceptions are thrown within the
code block, state "None."
4. **Variable Assignments**
   Given the variable list, provide the specific
 significance and role of the computed variable
in the code block in list form.
   If any variables are incorrectly identified
or unused in subsequent sections of code, these
can be directly removed.
   If the variable list is missing any modified
variable (such as ‘self.blockid_list.append(
block)‘), please add it to the list.
   Variable list: {variable_list}

### Sample Output:
1. **Purpose**
   Parse the target string to extract key
information. The target string is in the format
‘blocks = ["blockid1", "blockid2", ...]‘. This
code block extracts all valid blockids,
generating a new list of strings.
2. **Logic**
   Uses regular expressions (re library) to
extract blockid list from the target string,
then iterates the list, verifies each blockid's
existence in the database, and stores them
converted to integer type in a new list.
3. **Exceptions**
   - ‘ValueError‘: If the target string has an
incorrect format, making it unable to extract a
valid blockid list, this exception is thrown.
4. **Variable Assignments**
   - ‘self.blockid_list‘: Stores extracted and
validated blockids

### Code Block to be Analyzed:
{key_block}
```

Contextual Information of Code Block:
{class_code}

Prompt for Refinement

The code reviewers found the generated code explanation has the following issues:
{response}

Please modify the current code explanation based on the content of the code block and the reviewers' suggestions, and output it according to the specified format, **do not include extra content**.
Code Block to be Analyzed:
{key_block}

Current Code Explanation:
{explanation}

Output Requirements:
1. **Purpose**
   Describe the main goal of the code block and its role within the entire program. Specifically, what is its responsibility within the current function?
2. **Logic**
   Elaborate on the core logic and operational process of the code block. For all conditional branches (if statements), explain them one by one.
   If complex variable updates are involved, use Markdown format for formulas to represent these mathematical calculations.
   If variables from previous sections of the code block are used, try to describe using their variable names, enclosing them in backticks.
3. **Exceptions**
   If the analyzed code block throws exceptions (using ‘raise‘ statements, excluding ‘except‘ statements), explain its exceptional cases and types. If no exceptions are thrown within the code block, state "None."
4. **Variable Assignments**
   Using the provided variable list, describe the specific significance and role of the computed variable in the code block in list form.

   If there are any erroneously identified variables (e.g., those not used later in the code), you may directly remove these. If the variable list is missing any modified variable (such as ‘self.blockid_list.append(block)‘), please add it to the list.

Sample Output:
1. **Purpose**
   Parse the target string to extract key information. The target string format is ‘blocks = ["blockid1", "blockid2", ...]‘. This code block extracts all valid blockids and generates a new list of strings.
2. **Logic**
   Uses regular expressions (re library) to extract blockid list from the target string, then iterates the list, verifies each blockid's existence in the database, and stores them converted to integer type in a new list.
3. **Exceptions**
   - ‘ValueError‘: If the target string has an incorrect format making it impossible to extract a valid blockid list, this exception is thrown.
4. **Variable Assignments**
   - ‘self.blockid_list‘: Stores extracted and validated blockids.

Annotation Criteria

The evaluation consists of three aspects: Readability, Accuracy, and Completeness. Each aspect is scored on a three-level scale:

- **0 points:** Unusable, with obvious flaws.
- **1 point:** Minor flaws.
- **2 points:** Perfect, no flaws.

Problems with a total score of 5 or higher across the three aspects are considered qualified.

---

1. Readability Comments should be clear, concise, and easy to understand, enabling developers to quickly grasp the code's functionality and purpose.

- Comments are clear and easily readable by software engineers.
- Sentences are fluent, with no typographical errors.
- Formatting complies with Markdown standards.
- Comments express the code's functionality or requirements accurately using minimal wording, avoiding verbosity while ensuring clarity.

2. Accuracy Comments must faithfully reflect the behavior of the code, ensuring that the functionality implemented based on the comments matches the actual code behavior.

- Comments accurately describe the code's functionality, and the described logic matches the original code implementation.
- Important functional functions that need to be used are clearly indicated.
- Important variables that are modified (including class member variables) are listed accurately, with clear explanations.
- Utility function selection is correct; incorrect selection results in 0 points.
- Exception handling is correctly identified.

3. Completeness Comments should cover all critical aspects of the code, especially inputs, outputs, data structures, algorithms, and edge cases, without omitting any key content that affects correct understanding.

- Comments cover all key aspects of the code, without missing important context (such as inputs, outputs, data structures, algorithms, edge cases).
- Comments are directly related to the code's functionality and do not contain irrelevant or redundant information.
- Comments do not omit any branch logic or other elements that may affect correct understanding; omissions that lead to missing functionality are considered completeness issues.

Table 11: **Human Annotation Criteria.**