CodeFlowBench: A Multi-turn, Iterative Benchmark for Complex Code Generation

Sizhe Wang[†1,3], **Zhengren Wang**[†1,2], **Dongsheng Ma**[1]
Yongan Yu[1,4], **Rui Ling**[1], **Zhiyu Li**[*2], **Feiyu Xiong**[2], **Wentao Zhang**[*1,5,6]
[1]Peking University [2]Institute for Advanced Algorithms Research, Shanghai
[3]Shanghai University of Finance and Economics [4]McGill University [5]Zhongguancun Academy
[6]Beijing Key Laboratory of Data Intelligence and Security (Peking University)
{lizy,xiongfy}@iaar.ac.cn, wentao.zhang@pku.edu.cn

Dataset Code

Abstract

Modern software development demands code that is maintainable, testable, and scalable by organizing the implementation into modular components with iterative reuse of existing codes. We formalize this iterative, multi-turn paradigm as *codeflow* and introduce **CodeFlowBench**, the first benchmark designed to comprehensively evaluate LLMs' ability to perform codeflow - implementing new functionality by reusing existing functions over multiple turns. CodeFlowBench comprises two complementary components: CodeFlowBench-Comp, a core collection of 5,000+ competitive programming problems from Codeforces updated via an automated pipeline and CodeFlowBench-Repo, which is sourced from GitHub repositories to better reflect real-world scenarios. Furthermore, a novel evaluation framework featured dual assessment protocol and structural metrics derived from dependency trees is introduced. Extensive experiments reveal significant performance degradation in multi-turn codeflow scenarios. Furthermore, our in-depth analysis illustrates that model performance inversely correlates with dependency complexity. These findings not only highlight the critical challenges for supporting real-world workflows, but also establish CodeFlowBench as an essential tool for advancing code generation research.

1 Introduction

Large Language Models (LLMs) have revolutionized code generation, with benchmarks like HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) establishing foundational standards. As LLM capabilities advance, their role in real-world software development has expanded beyond solving toy problems to supporting complex workflows (Jiang et al., 2024; Jin et al., 2024; Liu et al., 2024b). Modern benchmarks such as SWE-Bench (Jimenez et al., 2023; Li et al., 2024) now emphasize practical scenarios like bug fixing. However, current benchmarks (Gu et al., 2024; Huang et al., 2024; Wang et al., 2025) still overlook the critical aspect of real-world development: the multi-turn and iterative *codeflow* scenario.

The CodeFlow Task In modern software engineering, workflows based on *iterative implementation* are becoming increasingly prevalent, as the cornerstones of best practices like agile development (Larman, 2004; Abrahamsson et al., 2017; Cram, 2019). Unlike *iterative refinement*, this paradigm requires breaking down complex tasks into manageable subproblems, progressively building solutions, and reusing modular functions. Through this sequential approach, developers can achieve faster delivery, reduced redundancy, and enhanced maintainability in teamwork (Haefliger et al., 2008; Feitosa et al., 2020). For example, React's core package alone sees over 37 million weekly downloads across 2,300+ dependent modules, illustrating the productivity gains of modular reuse (Boduch, 2019). As shown in Fig. 1 and 2, by building solutions via traversing the function dependency tree from the bottom up, the structured *codeflow* can both enables parallel development and improves readability, testability and maintainability through modular and responsibility boundaries. To integrate effectively into codeflows, LLMs must learn to "look before and after", namely accurately leveraging pre-implemented functions while ensuring the modularity of codes to generate.

Motivation Despite the growing demands, current benchmarks have not fully captured the multi-turn and iterative aspects of codeflow. Firstly, most benchmarks, such as HumanEval and MBPP, focus only on single-turn code generation. While recent benchmarks like BigCodeBench (Zhuo et al., 2024) and SWE-Bench (Jimenez et al., 2023) have be-

† Equal contribution; * Corresponding author.
The first author completed this work during an internship at Peking University.

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 4369–4402

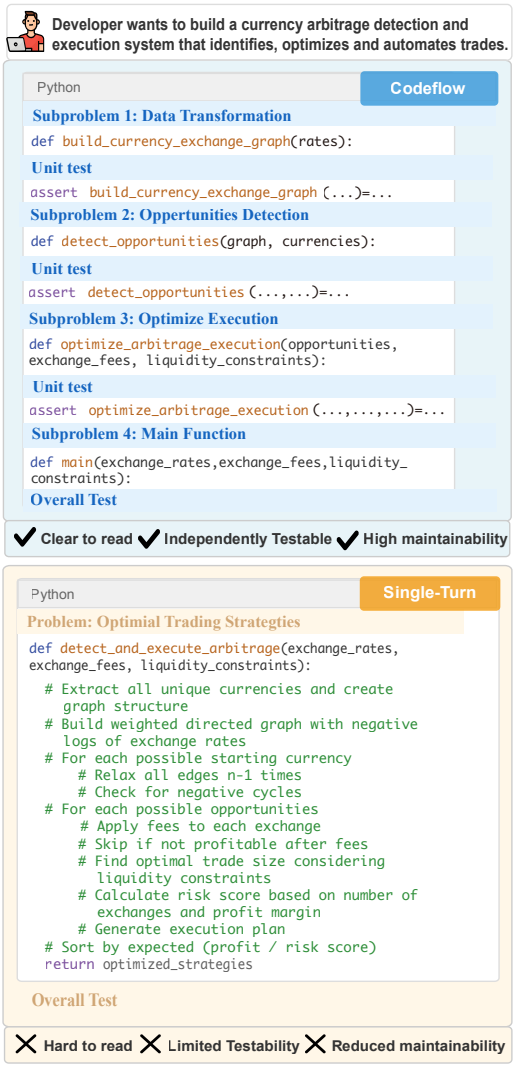


Figure 1: A currency arbitrage example contrasting the modular *codeflow* workflow (top) with a monolithic single-turn implementation (bottom).

gun to incorporate practical development scenarios, they still remain at the level of single-turn code modification (or two-turn code generation), leaving the multi-turn generation capabilities unknown. Secondly, the few existing multi-turn benchmarks such as MTPB (Nijkamp et al., 2022), only focus on single-function programming, lacking both unit tests and sufficient complexity to reflect real-world dependencies. Finally, due to the absence of an update mechanism, previous static datasets risk contamination and unreliable assessment. Therefore, there is a pressing need for a more challenging, well-annotated and frequently updated benchmark specially designed for codeflow.

To bridge this gap, we introduce CodeFlowBench, a benchmark suite that integrates two complementary datasets: *CodeFlowBench-Comp*, which is sourced from competitive programming platforms like Codeforces[1] and *CodeFlowBench-Repo*, which is derived from diverse real-world GitHub repositories via DomainEval (Zhu et al., 2025). Utilizing a unified automated pipeline, CodeFlowBench transforms raw data into sequences of multi-turn, iterative subproblems, accompanied by verified solutions and test cases. This dual-component architecture ensures a broad evaluative scope, spanning reasoning-intensive algorithmic challenge and labor-intensive software engineering. Specifically, *CodeFlowBench-Comp* provides high-difficulty, high-quality problems with periodic updates, while *CodeFlowBench-Repo* injects real-world complexity into the assessment. To further reveal model deficiencies, CodeFlowBench introduces a specialized evaluation framework and incorporates structural labels and metrics derived from dependency trees, enabling a nuanced analysis of multi-turn performance.

Contributions Our contributions are threefold:

- **Pipeline Innovation**: We developed a universal data curation pipeline grounded in function dependency analysis. This fully-automated and lightweight framework decomposes monolithic solutions into multi-turn, iterative coding problems that necessitate strategic code reuse. Its design is rigorously verifiable and highly extensible, enabling the transformation of diverse source code into structured codeflow-style challenges.
- **Benchmark Construction**: We introduce CodeFlowBench, the first benchmark to evaluate the iterative, multi-turn code generation capabilities of LLMs. By integrating two complementary datasets: *CodeFlowBench-Comp* (algorithmic depth) and *CodeFlowBench-Repo* (real-world domain breadth), we provide a comprehensive testbed to examine software engineering proficiency.
- **Evaluation Design & Insights**: We propose a novel evaluation framework that directly contrasts multi-turn and single-turn generation patterns. By introducing structural metrics, such as Average Pass Depth (APD) and Dependency Structure Complexity (DSC), derived from dependency trees, our framework captures the unique complexities of iterative

[1] https://codeforces.com/

tasks. Extensive experiments reveal a significant performance degradation in codeflow scenarios, even among state-of-the-art reasoning models, highlighting a critical frontier for future research and the need for more advanced codeflow programming capabilities.

2 Related Work

Code Generation Benchmarks The landscape of code generation benchmarks has evolved from simpler to more complex tasks to keep pace with the rapid development of LLMs (Li et al., 2022; Quan et al., 2025; Yu et al., 2024), but still fail to comprehensively capture the multi-turn and iterative features of real-world scenarios. Early works like HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) focus on standalone functions with low complexity and limited dependency environments. Recent benchmarks have emerged to evaluate more complex and realistic scenarios, yet have obvious limitations: most benchmarks such as APPS (Hendrycks et al., 2021), LiveCodeBench (Jain et al., 2024) and SWE-Bench (Jimenez et al., 2023) are limited to single-turn code generation or modification. For the few existing multi-turn benchmarks, MTPB (Nijkamp et al., 2022) focus on overly simplistic single-function programming without paired unit tests, while InterCode (Yang et al., 2023) discusses interactive coding with execution feedback. In stark contrast, *codeflow* structures the development into multi-turn processes, ensuring each component is maintainable, testable and reusable. These limitations highlight the crucial gaps for codeflow benchmarking, and the CodeFlowBench pioneers this research line.

Code Generation LLMs Recent years have witnessed unprecedented progress in code generation capabilities of LLMs. Early works such as Codex (Chen et al., 2021) and AlphaCode (Li et al., 2022) demonstrated proficiency in tasks ranging from code completion to competition-level problem solving. With the scaling up of pre-trained models, exemplified by GPT (OpenAI, 2025), Claude (Anthropic, 2025), Deepseek-Coder (Guo et al., 2024) and Qwen-Coder (Yang et al., 2025), these advanced models have impressive performance across various programming tasks, languages and domains. Building on these foundations, the code generation capabilities have further advanced through instruction tuning and agent frameworks. Models such as WizardCoder (Luo et al., 2023) and

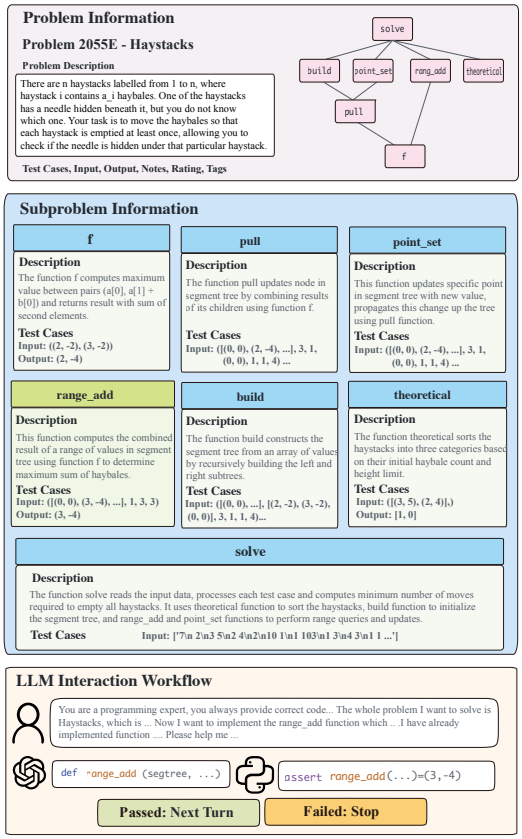


Figure 2: An illustrative example from CodeFlowBench (Source: Codeforces 2055E). The figure illustrates the problem definition with its dependency structure (top), the iterative subproblem decomposition (middle), and the corresponding LLM interaction workflow (bottom).

Magicoder (Wei et al., 2023) leverage instruction tuning to improve intent alignment and interactive dialogue capabilities, while agent frameworks like AgentCoder (Huang et al., 2023) and MapCoder (Islam et al., 2024) enable autonomous planning, iterative refinement, and self-evaluation. Despite advancements, the community still remains unknown about *"how well and how deeply LLMs can perform codeflow"*—a critical paradigm for real-world software engineering. Our CodeFlowBench thus provides a principled framework for advancing both model development and evaluation.

3 CodeFlowBench

In this section, we introduce CodeFlowBench's data curation pipeline and evaluation framework. Fig. 2 provides an example of CodeFlowBench.

3.1 Data Curation Pipeline

We designed an automated, lightweight data-curation pipeline to generate complex multi-turn

coding problems. As shown in Fig. 3, the pipeline mainly consists a data preparation phase and a subproblem generation phase.

Data Preparation Phase To address the lack of reasoning-intensive programming datasets tailored for codeflow scenarios, this phase collects programming challenges from Codeforces to construct *CodeFlowBench-Comp*. We first scrape problem metadata and official editorial content from the platform. To ensure data consistency, raw editorials are processed using DeepSeek-R1 (Guo et al., 2025) to generate modular, parsable solution code. These generated solutions undergo rigorous verification via the Codeforces submission system; only implementations that pass all official test cases are retained. By anchoring our dataset in official editorials and validating them against a live judge, we guarantee high-quality, correct codeflow implementations for every problem. Further specifics are provided in Appendix A.1–A.3.

Subproblem Generation Phase This phase functions as a *general framework* for transforming monolithic solutions into the codeflow format. First, we parse the AST of the verified solution to extract and topologically sort function dependencies. We treat each function as a distinct subproblem, employing Deepseek-V3 (Liu et al., 2024a) to back-translate its logic into a natural language statement. Next, we execute the full solution against public test cases, instrumenting function calls to capture input-output pairs as ground-truth unit tests for each subproblem. Finally, we annotate problem complexity such as dependency structure complexity, metrics that correlate strongly with official difficulty ratings. Implementation details are provided in Appendix A.4–A.5.

Remark. Our AST-based decomposition pipeline is fundamentally grounded in established software engineering principles. By decomposing monolithic solutions at the function level via AST parsing, our methodology inherently enforces the Single Responsibility Principle (SRP) and Separation of Concerns (SoC). Each extracted subproblem represents a cohesive logical unit with well-defined interfaces, ensuring the benchmark evaluates realistic modularity. Moreover, our strict source curation prevents the inheritance of underlying structural flaws. To further ensure this pipeline is not biased by the LLMs used for subproblem description and code generation, we conducted several ablation studies. We assessed description consistency across models, validated quality via Human-LLM cross-validation, and verified the structural stability of code decomposition across different LLMs. Ablation details are found in Appendix A.6, and cost estimates for pipeline running are in Appendix A.7.

3.2 Benchmark Details

CodeFlowBench-Comp This component comprises 5,258 competitive programming problems designed to test algorithmic reasoning depth. It emphasizes multi-step logic with up to $\geq$7 interaction turns. Similarly, dependency structures are non-trivial, concentrated at depth 2, indicating that nested function calls are the norm rather than the exception. To validate the difficulty of these structures, we correlated our metrics with official Codeforces ratings. Fig. 4 demonstrates the statistical information of CodeFlowBench-Comp. Regarding the distribution, the data shows a clear preference for multi-step logic. Furthermore, correlating these metrics with Codeforces difficulty ratings confirms their validity. As illustrated in the rating curves, we observe a strong positive correlation beyond the structural baseline: as the interaction turns and dependency depth increase, the average problem rating rises sharply—approaching 2,900 for tasks with $\geq$7 turns. This demonstrates that our structural metrics effectively capture the intrinsic complexity of logical problems.

CodeFlowBench-Repo Complementing the algorithmic focus, CodeFlowBench-Repo evaluates software engineering proficiency in real-world contexts. It is constructed by applying our Subproblem Generation pipeline to DomainEval (Zhu et al., 2025), utilizing high-quality GitHub repositories ($>$100 stars) as the source. Our curated dataset spans five specialized domains: *Basic, Cryptography, Network, System, and Visualization*. Through our pipeline, we extended these original contexts into complex codeflow scenarios that reach up to 9 interaction turns and a dependency depth of 4. The final distribution highlights significant structural complexity: 56.6% of problems require multi-turn iterative generation, forcing models to maintain reasoning continuity, while 54.3% involve deep dependency chains, requiring the model to correctly orchestrate hierarchical function calls. Detailed construction protocols and statistics are provided in Appendix B.1, B.2 and B.3.

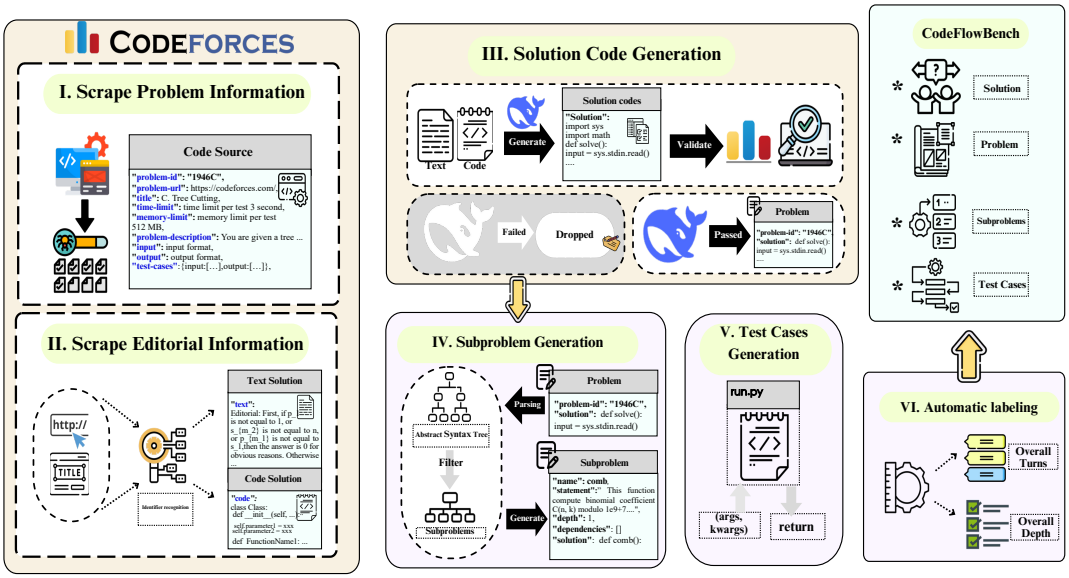
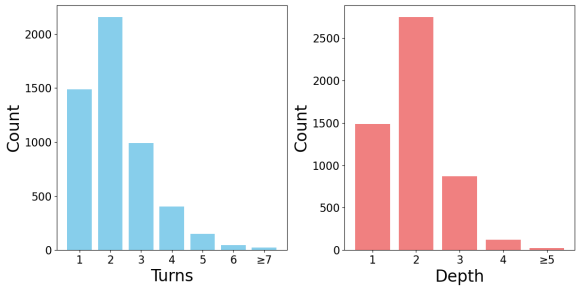
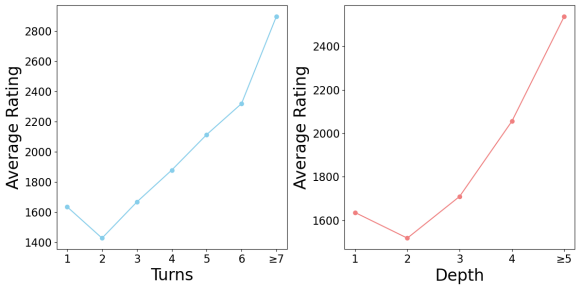


Figure 3: The data curation pipeline of CodeFlowBench. In **Data Preparation Phase** (Steps I–III), we scrape problem statements and editorial information, followed by generating and validating the canonical solution codes. In **Subproblem Generation Phase** (Steps IV–VI), we decompose solutions into subproblems via AST parsing, generate corresponding test cases, and perform automatic complexity labeling to achieve the final CodeFlowBench.



(a) Distributions of overall-turns and overall-depth.



(b) The Correlations with Rating Levels.

Figure 4: Statistics of the overall-turns and overall-depth metrics in CodeFlowBench-Comp. Subfigure (b) shows inflection points at turns = 1 and depth = 1. This is attributed to the fact that competition-level problems are not restricted to multi-turn or deeply nested structures. Partial difficult problems are designed to be solvable by single function such as number theory related problems.

3.3 Evaluation Framework

Task Definition Our evaluation follows best practices defined in HumanEval (Chen et al., 2021), where models are required to implement a given function in each round. However, CodeFlowBench introduces distinct differences in the supporting materials provided to the models. We include the function signature F_t, problem description S_t and pre-implemented functions for code reuse. Additionally, we provide an overarching background description of the problem as B. This reflects the real-world scenario where developers often possess overall understanding of the entire problem when implementing code incrementally. As illustrated in Fig. 2, models must implement target functions, C_t at turn t, by leveraging pre-implemented components from prior turns $\{C_1, ..., C_{t-1}\}$.

For baseline comparison, we define the single-turn variant where the model generates all components simultaneously. The mathematical expressions of both settings are presented below:

$$C_{1:T} \leftarrow p\Big(\cdot \mid F_{1:T}, S_{1:T}, B\Big).$$
$$C_t \leftarrow p\Big(\cdot \mid F_t, S_t, C_{<t}, B\Big), \quad 1 \le t \le T.$$

Performance Metrics We adopt widely-used Pass@k (Chen et al., 2021) as the main metric for both multi-turn and single-turn cases, but aug-

ment it with novel diagnostics for multi-turn analysis. While Pass@k evaluates final success rates, the coarse-grained nature fails to capture partial progress in failed attempts -two models may fail solve the entire problem at different stages but receive identical scores.

To address this limitation, we propose a new metric, Pass Turn (PT), which identifies the exact turn at which a model fails by leveraging unit tests for each subproblem. However, only considering failing turns may be biased due to the arbitrary topological ordering at the same depth. We further define, Pass Depth ($PD = D - d$), for bottom-up programming, where d and D are working depth and total depth. For statistical significance, we define the Average Pass Turn (APT) and Average Pass Depth (APD) by averaging across problems grouped by turn or depth. For multiple trials, we define APD@k and APT@k following the Pass@k expression. These metrics clearly distinguishing a model's strong performance on shallow steps from its potential struggles with deep dependencies. Please refer to Appendix C for more details.

4 Experiments

Experiment Setup For comprehensive experiments, we evaluate both close-sourced and open-sourced models. The closed-source models include the GPT (OpenAI, 2025), Gemini (Google, 2025) and Claude families (Anthropic, 2025). The open-source models include the Qwen (Yang et al., 2025), DeepSeek (Liu et al., 2024a; Guo et al., 2025) and Llama families (Grattafiori et al., 2024). We experiments on *CodeFlowBench-Comp* and *CodeFlowBench-Repo* under identical settings. For *CodeFlowBench-Comp*, we selected a test subset comprising the most recent 1,000 problems to eliminate potential data leakage and decrease evaluation overhead while maintaining statistical significance. All models are evaluated in both multi-turn and single-turn scenarios, utilizing Pass@k and APD@k as the primary metrics. Please refer to Appendix D.1 and D.2 for implementation details and inference hyperparameter.

4.1 Main Experiments

Table 1 presents the main experiment results. The statistic rigor of evaluation result is reported in Appendix D.3. For the majority of LLMs, the Pass@1 in the multi-turn scenario remains below 30%, underscoring CodeFlowBench's difficulty. The performance across CodeFlowBench also reveals divergent model specializations. For example, Gemini-3-flash performs best on CodeFlowBench-Comp while GPT-5 outperforms on CodeFlowBench-Repo. Furthermore, we present more analysis as follows.

Multi-Turn versus Single-Turn Table 1 reveals a substantial performance gap between multi-turn and single-turn scenarios across both *CodeFlowBench-Comp* and *CodeFlowBench-Repo*. This consistent degradation underscores the inherent complexity of multi-turn generation, which necessitates long-context coherence and robust dependency management. Notably, we observe a divergence in model behaviors: while reasoning-enhanced models like Deepseek-R1 and o3-mini achieve impressive single-turn scores, they suffer performance drops exceeding 50% in multi-turn settings. In contrast, models such as GPT-5 demonstrate greater stability with narrower gaps. This distinction suggests that single-turn metrics largely reflect peak reasoning capabilities, whereas multi-turn evaluation exposes previously under-explored limitations in iterative development and dependency management.

Depth-Wise Performance Our analysis across dependency depths reveals a dichotomy in scaling laws. First, in shallow dependencies, reasoning paradigms supersede parameter scale. Efficient reasoning models, such as o3-mini, significantly outperform larger base models like GPT-4o on *CodeFlowBench-Comp*, suggesting that logical deduction is more critical than raw capacity for immediate or localized tasks. Second, deep dependencies function as a **capacity filter**, where model scale becomes decisive. In *CodeFlowBench-Repo*, Qwen2.5-72B-Instruct doubles the performance of its 7B and 32B counterparts, indicating that maintaining coherence across extended dependency chains requires substantial parameter capacity. Finally, SOTA models like Gemini-3-flash-thinking and GPT-5 demonstrate full-spectrum dominance, eliminating trade-offs by combining agile reasoning for local logic with the extensive capacity required for global integration.From another perspective, we report APT@1 in Table 2. Notably, the APT trends closely align with APD, reinforcing our observations of models' depth-wise performance.

Furthermore, we conduct extensive ablation studies to validate our evaluation framework. Specifi-

Model	Pass@1 (%)		APD@1 (Average Pass Depth)				
	Multi-Turn	Single-Turn	Overall	Depth1	Depth2	Depth3	Depth4
CODEFLOWBENCH-COMP							
Closed-Source							
GPT-5	26.5	37.6	0.762	0.185	0.866	1.306	1.542
GPT-5-mini	19.1	38.0	0.655	0.103	0.748	1.200	1.375
o3-mini	22.7	38.9	0.570	0.322	0.585	0.818	1.250
o1-mini	20.8	37.8	0.541	0.233	0.581	0.818	1.125
GPT-4.1-mini	24.4	38.7	0.602	0.265	0.673	0.873	1.042
GPT-4o-mini	13.8	22.0	0.423	0.138	0.438	0.697	1.167
Gemini-2.5-flash-thinking	22.7	39.8	0.642	0.167	0.761	0.981	1.208
Gemini-3-flash-thinking	**48.4**	**65.5**	**1.120**	**0.445**	**1.303**	**1.625**	**1.625**
Claude-4-Sonnet	27.3	42.9	0.830	0.185	0.893	1.250	1.375
Claude-4.5-Sonnet	31.6	47.9	0.857	0.222	1.008	1.377	1.521
Open-Source (7B)							
Llama-3.1-8B-Instruct	0.9	3.5	0.208	0.011	0.224	0.412	0.792
Qwen2.5-Coder-7B-Instruct	2.3	15.0	0.233	0.018	0.247	0.436	0.750
Qwen3-8B	**9.1**	**18.3**	**0.406**	**0.088**	**0.444**	**0.738**	**1.125**
Open-Source (32B)							
Qwen2.5-Coder-32B-Instruct	8.6	19.8	0.316	0.067	0.342	0.570	**0.917**
Qwen3-Coder-30B-A3B	**11.6**	**26.0**	**0.405**	**0.100**	**0.458**	**0.681**	**0.917**
Open-Source (70B)							
Llama-3.3-70B-Instruct	**15.0**	**27.6**	**0.448**	**0.163**	**0.465**	**0.733**	**1.042**
Qwen2.5-72B-Instruct	9.1	21.3	0.301	0.110	0.314	0.497	0.583
Open-Source (Large)							
Deepseek-V3	18.0	35.7	0.529	0.219	0.549	0.836	**1.208**
Deepseek-R1	**20.5**	**46.1**	**0.569**	**0.303**	**0.606**	**0.842**	0.916
CODEFLOWBENCH-REPO							
Closed-Source							
GPT-5	**34.7**	48.8	0.898	0.483	1.020	1.385	3.200
GPT-5-mini	28.1	53.5	0.740	0.362	0.725	1.615	3.000
o3-mini	28.3	51.2	0.850	0.466	0.902	1.615	2.800
o1-mini	21.4	48.8	0.817	0.362	0.860	1.615	**3.600**
GPT-4.1-mini	27.6	48.0	0.724	0.431	0.705	**1.692**	1.800
GPT-4o-mini	24.6	43.3	0.667	0.414	0.800	1.000	1.400
Gemini-2.5-flash-thinking	26.1	44.6	0.765	0.375	0.833	1.167	**3.600**
Gemini-3-flash-thinking	33.9	56.7	0.905	**0.500**	1.020	1.308	3.400
Claude-4-Sonnet	33.9	**57.6**	0.850	0.461	0.980	1.385	2.600
Claude-4.5-Sonnet	33.1	56.7	**0.913**	0.448	**1.039**	1.615	3.200
Open-Source (7B)							
Llama-3.1-8B-Instruct	17.3	39.6	0.417	**0.310**	0.451	0.615	0.800
Qwen2.5-Coder-7B-Instruct	**19.7**	**39.1**	**0.528**	**0.310**	**0.647**	**0.692**	**1.400**
Qwen3-8B	18.1	36.2	0.394	0.293	0.450	0.462	0.800
Open-Source (32B)							
Qwen2.5-Coder-32B-Instruct	**25.2**	**45.7**	**0.646**	0.379	**0.784**	1.000	**1.400**
Qwen3-Coder-30B-A3B	23.6	42.5	0.543	**0.414**	0.470	**1.385**	0.600
Open-Source (70B)							
Llama-3.3-70B-Instruct	22.1	39.6	0.614	0.345	0.522	**1.846**	1.400
Qwen2.5-72B-Instruct	**25.2**	**43.3**	**0.716**	**0.397**	**0.745**	1.230	**2.800**
Open-Source (Large)							
Deepseek-V3	29.1	45.1	**0.771**	**0.483**	0.745	**1.538**	2.400
Deepseek-R1	**29.4**	**52.0**	0.770	0.431	**0.760**	**1.538**	**2.800**

Table 1: Performance on CodeFlowBench-Comp and CodeFlowBench-Repo across different model scales. Both datasets report Pass@1 and APD@1 (Average Pass Depth) across varying depths.

cally, we analyze: (1) the impact of the multi-turn paradigm versus monolithic generation; (2) the effectiveness of our data contamination mitigation; (3) the effect of the "fail-stop" mechanism; and (4) model sensitivity to different topological orderings of subproblems. Detailed results are provided in Appendix E.

4.2 Analysis and Discussion

Dependency Structure Challenges in Multi-turn Scenarios A deeper analysis of solved problems reveals a striking imbalance: the majority of correctly addressed cases correspond to problems with simple, linear dependency structures (e.g., shallow call graphs or sequential compositions). However, as problem architectures evolve toward modular and hierarchical dependencies (e.g., nested function calls, interdependent components), models exhibit significant performance degradation. This phenomenon is empirically validated in Figure 5, which illustrates the Pass@1 scores across varying turn counts. The consistent performance trajectory demonstrates the inherent challenges in multi-turn code generation, where curves initially high for 1-2 turn problems and followed by sharp decline as turn counts increase. Even top-performing models fail to solve problems requiring more than six turns. This underscores the critical limitation to balance

Model	CodeFlowBench-Comp					CodeFlowBench-Repo			
	Overall	Turn 1	Turn 2	Turn 3	Turn 4	Overall	Turn 1	Turn 2	Turn 3+
Closed-Source									
GPT-5	0.832	0.185	0.951	1.224	1.781	0.890	**0.509**	1.049	1.000
GPT-5-mini	0.717	0.103	0.832	1.158	1.345	0.520	0.363	0.537	0.667
o3-mini	0.600	0.322	0.632	0.777	0.983	0.780	0.455	0.854	1.056
o1-mini	0.581	0.233	0.645	0.798	0.879	0.675	0.345	0.725	0.944
GPT-4.1-mini	0.646	0.265	0.760	0.803	1.034	0.701	0.436	0.732	1.056
GPT-4o-mini	0.467	0.137	0.501	0.638	0.982	0.571	0.418	0.650	0.778
Gemini-2.5-flash-thinking	0.695	0.167	0.855	0.923	1.218	0.687	0.364	0.879	0.600
Gemini-3-flash-thinking	**1.178**	**0.445**	**1.386**	**1.617**	**1.800**	0.905	0.491	1.000	1.000
Claude-4-Sonnet	0.830	0.185	0.990	1.153	1.564	0.866	0.455	1.024	**1.389**
Claude-4.5-Sonnet	0.927	0.222	1.104	1.291	1.704	**0.961**	0.418	**1.146**	1.167
Open-Source (7B)									
Llama-3.1-8B-Instruct	0.232	0.011	0.245	0.404	0.534	0.370	**0.327**	0.439	0.389
Qwen2.5-Coder-7B-Instruct	0.258	0.018	0.270	0.394	0.638	**0.528**	**0.327**	**0.512**	**0.778**
Qwen3-8B	**0.452**	**0.075**	**0.490**	**0.692**	**1.036**	0.362	0.309	0.488	0.333
Open-Source (32B)									
Qwen2.5-Coder-32B-Instruct	0.352	0.067	0.391	0.532	0.569	**0.583**	0.364	**0.780**	**0.833**
Qwen3-Coder-30B-A3B	**0.449**	**0.100**	**0.508**	**0.699**	**0.836**	0.496	**0.418**	0.561	0.333
Open-Source (70B)									
Llama-3.3-70B-Instruct	**0.493**	**0.163**	**0.515**	**0.681**	**1.000**	0.516	0.345	0.553	0.750
Qwen2.5-72B-Instruct	0.330	0.110	0.350	0.452	0.517	**0.669**	**0.364**	**0.780**	**0.889**
Open-Source (Large)									
Deepseek-V3	0.572	0.219	0.622	0.750	**0.966**	**0.709**	**0.455**	0.829	**0.722**
Deepseek-R1	**0.609**	**0.304**	**0.677**	**0.766**	**0.966**	0.675	0.418	**0.925**	0.611

Table 2: Pass Turn comparison across models on CodeFlowBench-Comp and CodeFlowBench-Repo. Consistent with Table 1, the overall performance distribution shows that models with high APD@1 scores also exhibit high APT@1 scores, validating the precision of our metric.

local correctness and global integration across iterative development cycles.

To quantity dependency structure complexity, we introduce the *Dependency Structure Complexity (DSC)* metric, defined as the ratio of total turns to the maximum depth in the AST. Figure 6 presents the models' performance across different DSC intervals, revealing that most models perform well on problems with linear dependency structures but struggle significantly as the dependency structure becomes more complex. For a problem, the DSC metric is defined as:

$$\text{DSC(problem)} = \frac{\text{Overall-Turns(problem)}}{\text{Overall-Depth(problem)}}$$

Recall that the overall-turn and overall-depth of a problem are derived from its AST, corresponding to the number of nodes and the depth of its AST. Based on this, we can see that a high *DSC* value indicates a problem with a complex dependency structure. Figure 6 presents the *pass@1* scores of models across different *DSC* intervals. It can be observed that most models are only capable of solving problems with *DSC* equal to 1, which corresponds to a simple linear dependency structure. Only a few leading models are able to solve a limited number of problems with *DSC* values below 1.33. All models struggle significantly when faced with problems involving more complex structures.

Fine-Grained Error Types in Multi-turn Generation Given the significant performance gap between models in multi-turn and single-turn scenarios, we conducted studies to identify the underlying reasons. We categorized errors into three primary types: (1) **Incomplete Reasoning (IR):** Models often handle only straightforward "happy-path" cases and fail to generalize. They may oversimplify key requirements, omit boundary or atypical cases, or choose naive algorithms whose logic or performance collapses on larger inputs. This reflects a limitation in the models' reasoning abilities. (2) **Insufficient Globalization (IG):** While a function's logic may run correctly in isolation, it may omit necessary imports, global constants, or shared-state interactions, preventing proper integration into the broader application or runtime. This indicates a limitation in the models' ability to manage global context. (3) **Instruction Misinterpretation (IM):** Given multi-turn prompts, models could solve isolated subproblems but lack a coherent understanding of the overarching goal. Typical failures include misusing helper functions or implementing disorganized code within the top-level function, i.e. incorrect integration of components. To quantify the distribution of errors, we randomly sampled examples from *CodeFlowBench-Comp* and *CodeFlowBench-Repo*. The error categories are manually annotated by authors. The proportion is calculated in Table 3.

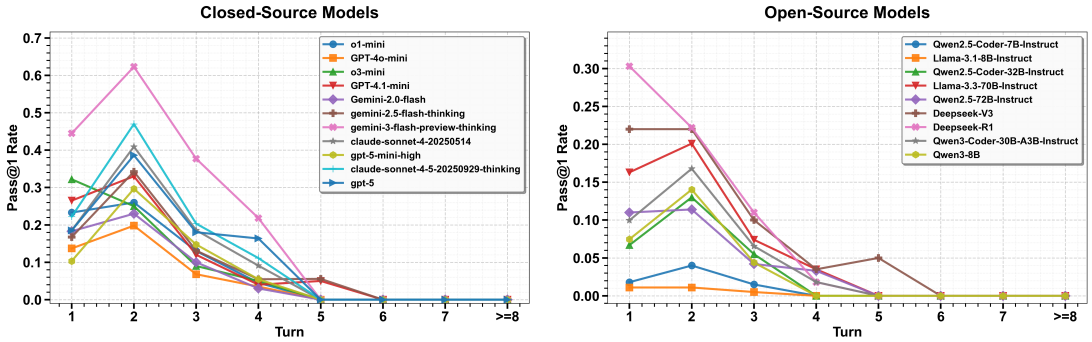


Figure 5: Models' Pass@1 results on multi-turn problems grouped by model categories and turn number.

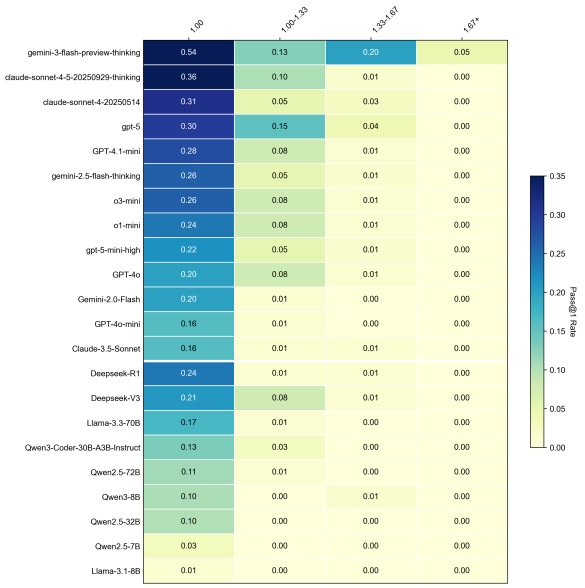


Figure 6: Heatmap of models' pass@1 scores on multi-turn problems within different DSC intervals.

Table 3 offers several insights. First, the distinct error distributions confirm that our benchmark components target divergent primary challenges: *CodeFlowBench-Comp* is dominated by IR and *CodeFlowBench-Repo* emphasizes global context which shifting the burden to IG and IM. Second, distinct bottlenecks constrain models of different tiers. DeepSeek-V3 and o1-mini are primarily limited by IM, suggesting their performance is capped at instruction comprehension and adherence. In contrast, for SOTA models like Gemini-3-Flash, the bottleneck shifts back to IR, indicating task understanding becomes minor issues. Detailed case studies are provided in Appendix E.5.

5 Conclusion

This paper introduces CodeFlowBench, the first benchmark specifically designed to evaluate multi-turn iterative code generation capabilities in realistic development workflows, i.e., the *codeflow*. CodeFlowBench consists of two subsets: CodeFlowBench-Comp for multi-step algorithmic reasoning via competition challenges, and CodeFlowBench-Repo for evaluating generalization in real-world software contexts. Our benchmark makes three key contributions: (1) an automated pipeline for decomposing complex problems into dependency-aware subproblems with paired unit tests, (2) a novel evaluation framework with proposed structural metrics, such as APD@k and DSC, to quantify multi-turn performance, and (3) the discovery of substantial performance gaps between multi-turn and single-turn scenarios (up to 50% performance degradation). Our fine-grained analysis identifies dominant failure modes and provides insights for further advancements. Extensive experiments across 19 popular LLMs highlight the substantial challenges posed by both the codeflow task and our benchmark. We believe CodeFlowBench not only illuminates critical limitations in existing LLMs but also paves the way for more realistic and powerful code generation systems.

Model	Dataset	IR(%)	IG(%)	IM(%)	Others(%)
DeepSeek-V3	Comp	47.2	14.6	32.6	5.6
	Repo	32.1	22.6	44.0	1.2
o1-mini	Comp	38.7	11.8	43.0	6.5
	Repo	23.7	18.3	55.9	2.2
Gemini-3-Flash	Comp	52.3	20.5	27.3	0.0
	Repo	34.6	21.8	41.0	2.6

Table 3: Error distribution across models and benchmarks. Values indicate the percentage of each error category relative to the total number of errors identified in the sampled dataset for each model.

Limitations

We acknowledge two limitations that present opportunities for future expansion. First, while the pipeline's subproblem generation phase is general, the data preparation phase is currently specific. To achieve universal compatibility, we can consider implementing an agentic framework for automatically adapting diverse codebases. This will expand our pipeline's coverage to a broader range of data sources beyond Codeforces and GitHub, such as Bitbucket, establishing a truly source-agnostic, end-to-end framework. Second, we currently lack models specifically fine-tuned for the codeflow task. Future work will consider fine-tuning specialized models and agentic systems on this data to assist developers in real-world, iterative programming scenarios. We also aim to extend our codeflow paradigm to include iterative debugging loops. By training specialized systems on these dynamic scenarios, we seek to enable robust and sustainable code implementation and maintenance, allowing models to effectively resolve requirements through multi-turn refinement.

Ethical Consideration

Codeforces provides an official "Codeforces materials usage license"[2]. We have taken great care to make our pipeline fully comply with all licensing requirements and community guidelines:

Firstly, all problems we use are officially published by Codeforces, and we provide clear attribution to Codeforces. Our usage is strictly non-commercial and academic, in accordance with the license's permitted use cases. Our data pipeline strictly avoids scraping any user-submitted code, forum discussion content or third-party solutions. We only collect officially published materials, namely problem statements and editorials.

Secondly, officially published Codeforces materials have long been used for widely recognized code generation benchmarks such as CodeContest (Li et al., 2022), LiveCodeBench (Jain et al., 2024) and CodeElo (Quan et al., 2025). Our work follows this community tradition and with the utmost respect for Codeforces.

Besides, we will also make our code properly hosted and clearly state its intended use to ensure responsible data collection.

[2] https://codeforces.com/blog/entry/967

Acknowledgements

This work is supported by Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (JYB2025XDXM113), National Natural Science Foundation of China (92470121, 62402016), National Key R&D Program of China (2024YFA1014003), Zhongguancun Academy (C20250204, C20250602), Beijing Major Science and Technology Project (Z251100008125043, Z251100008425023), and High-performance Computing Platform of Peking University.

References

Pekka Abrahamsson, Outi Salo, Jussi Ronkainen, and Juhani Warsta. 2017. Agile software development methods: Review and analysis. *arXiv preprint arXiv:1709.08439*.

Anthropic. 2025. Claude-4 system card. System card, Anthropic.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Adam Boduch. 2019. *React material-ui cookbook: build captivating user experiences using react and material-ui*. Packt Publishing Ltd.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

W Alec Cram. 2019. Agile development in practice: Lessons from the trenches. *Information Systems Management*, 36(1):2–14.

Daniel Feitosa, Apostolos Ampatzoglou, Antonios Gkortzis, Stamatia Bibi, and Alexander Chatzigeorgiou. 2020. Code reuse in practice: Benefiting or harming technical debt. *Journal of Systems and Software*, 167:110618.

Google. 2025. Gemini-3-flash system card. System card, Google.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I Wang.

2024. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, and 1 others. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Stefan Haefliger, Georg Von Krogh, and Sebastian Spaeth. 2008. Code reuse in open source software. *Management science*, 54(1):180–193.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and 1 others. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

Dong Huang, Yuhao Qing, Weiyi Shang, Heming Cui, and Jie Zhang. 2024. Effibench: Benchmarking the efficiency of automatically generated code. *Advances in Neural Information Processing Systems*, 37:11506–11544.

Dong Huang, Jie M Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. 2023. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*.

Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. Mapcoder: Multi-agent code generation for competitive problem solving. *arXiv preprint arXiv:2405.11403*.

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*.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*.

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

Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. 2024. From llms to llm-based agents for software engineering: A survey of current, challenges and future. *arXiv preprint arXiv:2408.02479*.

Craig Larman. 2004. *Agile and iterative development: a manager's guide*. Addison-Wesley Professional.

Bowen Li, Wenhan Wu, Ziwei Tang, Lin Shi, John Yang, Jinyang Li, Shunyu Yao, Chen Qian, Binyuan Hui, Qicheng Zhang, and 1 others. 2024. Devbench: A comprehensive benchmark for software development. *CoRR*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, and 1 others. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, and 1 others. 2024a. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. 2024b. Large language model-based agents for software engineering: A survey. *arXiv preprint arXiv:2409.02977*.

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

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

OpenAI. 2025. Gpt-5 system card. System card, OpenAI.

Shanghaoran Quan, Jiaxi Yang, Bowen Yu, Bo Zheng, Dayiheng Liu, An Yang, Xuancheng Ren, Bofei Gao, Yibo Miao, Yunlong Feng, and 1 others. 2025. Codeelo: Benchmarking competition-level code generation of llms with human-comparable elo ratings. *arXiv preprint arXiv:2501.01257*.

Zhengren Wang, Rui Ling, Chufan Wang, Yongan Yu, Zhiyu Li, Feiyu Xiong, and Wentao Zhang. 2025. Maintaincoder: Maintainable code generation under dynamic requirements. *arXiv preprint arXiv:2503.24260*.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Empowering code generation with oss-instruct. *arXiv preprint arXiv:2312.02120*.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. 2023. Intercode: Standardizing and benchmarking interactive coding with execution feedback. *Advances in Neural Information Processing Systems*, 36:23826–23854.

Zhaojian Yu, Yilun Zhao, Arman Cohan, and Xiao-Ping Zhang. 2024. Humaneval pro and mbpp pro: Evaluating large language models on self-invoking code generation. *arXiv preprint arXiv:2412.21199*.

Qiming Zhu, Jialun Cao, Yaojie Lu, Hongyu Lin, Xianpei Han, Le Sun, and Shing-Chi Cheung. 2025. Domaineval: An auto-constructed benchmark for multi-domain code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 26148–26156.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, and 1 others. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*.

Appendix

A Data Curation Pipeline **13**
- A.1 Detail of Problem Scraping 13
- A.2 Detail of Solution Scraping 13
- A.3 Detail of Solution Code Generation 16
- A.4 Detail of Subproblems Generation 16
- A.5 Detail of Test Cases Generation . 16
- A.6 Ablation Study of LLM usage . . 16
- A.7 Compute Cost Estimation 22

B CodeFlowBench-Repo Dataset **22**
- B.1 DomainEval 22
- B.2 Dataset Construction 23
- B.3 Dataset Statistics 23

C Mathematical Expression of $PD@k$& $APD@k$ **23**

D Experiment **26**
- D.1 Experiment Setting 26
- D.2 Instruction Templates 26
- D.3 Statistic Rigor 27

E Ablation Study **27**
- E.1 Empirical Validation of Software Quality 27
- E.2 Analysis of Data Contamination Risks 29
- E.3 Analysis of Evaluation Protocol: Fail-Stop vs. Non-Stop 29
- E.4 Sensitivity to Topological Ordering 30
- E.5 Error Case Study 30

F Impact Statement **30**

A Data Curation Pipeline

A.1 Detail of Problem Scraping

The original problem contain two parts. The first part is scraped from its corresponding codeforces official website. An example problem page is shown in Figure 7.

The second part is scraped by *Problemsets.Problems*[3] API provided by Codeforces.We record the rating and tags of each problem. Rating is a metric that reflect the difficulty of the problem and tags is a list that contains knowledge scope of the problem.

[3] https://codeforces.com/api/problemset.problems

Combine the above two part, we obtain a original coding problem for CodeFlowBench, a full example is shown in Figure 8.

A.2 Detail of Solution Scraping

During the problem-scraping process, we also collected the links under the "Contest material" section on the right side of each problem page and identified which of those led to solution blogs. Crawling these editorial pages is fairly complex, because Codeforces' official write-ups are hosted as personal blogs whose formats can vary over time, making content extraction more challenging. Although each round's problems (e.g. "123A," "123B," etc.) live on separate pages, all of the editorials for a given numeric ID usually appear on a single blog page. We therefore need to assign each sub-problem's write-up (A, B, C, ...) to its corresponding problem. To do this, our crawler first locates the distinct anchor points that mark each sub-problem section, then extracts the content between each anchor and the next as that problem's editorial. The benefit of this approach is that, while in the end we only need the solution code itself, editorials often consist of plain text explanations, a single code snippet, or multiple code variants and languages. Relying on a purely mechanical scraper makes it difficult to isolate exactly the code we want, so it's more effective to pass the raw editorial content to an LLM for final organization and extraction.

In general, there are two ways to locate an anchor: by URL and by text. Blogs label problems in many different forms. However, most blogs make that label into a hyperlink pointing back to the original problem, which gives us a reliable way to identify the anchor.

Therefore, our primary and most precise method is to search for a URL containing the problem ID (for example, "problemset/problem/2060/A" or "contest/blog/2060/A") and treat its position as the anchor. Once the anchor is found, we scan the surrounding page and extract its content to obtain the editorial.Anchor scanning and recording also relies on a problem-ID reference table, which we built from the IDs of all problems scraped in the first step. Its main role is to guide the code when matching anchors: for instance, if the table shows that numeric ID 2060 has sub-IDs "A" through "G", the scraper first reads those sub-IDs, then walks through the page using the URL-based or text-based method to record the exact anchor for each sub-ID.An exam-

E. Opening Portals

time limit per test: 2 seconds
memory limit per test: 256 megabytes

Pavel plays a famous computer game. A player is responsible for a whole country and he can travel there freely, complete quests and earn experience.

This country has n cities connected by m bidirectional roads of different lengths so that it is possible to get from any city to any other one. There are portals in k of these cities. At the beginning of the game all portals are closed. When a player visits a portal city, the portal opens. Strange as it is, one can teleport from an open portal to an open one. The teleportation takes no time and that enables the player to travel quickly between rather remote regions of the country.

At the beginning of the game Pavel is in city number 1. He wants to open all portals as quickly as possible. How much time will he need for that?

Input

The first line contains two space-separated integers n and m ($1 \le n \le 10^5$, $0 \le m \le 10^5$) that show how many cities and roads are in the game.

Each of the next m lines contains the description of a road as three space-separated integers x_i, y_i, w_i ($1 \le x_i, y_i \le n$, $x_i \ne y_i$, $1 \le w_i \le 10^9$) — the numbers of the cities connected by the i-th road and the time needed to go from one city to the other one by this road. Any two cities are connected by no more than one road. It is guaranteed that we can get from any city to any other one, moving along the roads of the country.

The next line contains integer k ($1 \le k \le n$) — the number of portals.

The next line contains k space-separated integers $p_1, p_2, ..., p_k$ — numbers of the cities with installed portals. Each city has no more than one portal.

Output

Print a single number — the minimum time a player needs to open all portals.

Please, do not use the `%lld` specifier to read or write 64-bit integers in C++. It is preferred to use the `cin`, `cout` streams or the `%I64d` specifier.

Examples

input

```
3 3
1 2 1
1 3 1
2 3 1
3
1 2 3
```

output

```
2
```

input

```
4 3
1 2 1
2 3 5
2 4 10
3
2 3 4
```

output

```
16
```

input

```
4 3
1 2 1000000000
2 3 1000000000
3 4 1000000000
4
1 2 3 4
```

output

```
3000000000
```

Note

In the second sample the player has to come to city 2, open a portal there, then go to city 3, open a portal there, teleport back to city 2 and finally finish the journey in city 4.

Figure 7: An example page of problems on Codeforces, which contains *problem ID, title, time and memory complexity limits, problem description, input description, output description, sample tests, notes* of each problem.The original problem is 196E

.

"problem-id": "1946E",
"problem-url": "https://codeforces.com/problemset/problem/1946/E",
"title": "E. Girl Permutation",
"time-limit": "time limit per test 2 seconds",
"memory-limit": "memory limit per test 256 megabytes",
"problem-description": "Some permutation of length n is guessed. You are given the indices of its prefix maximums and suffix maximums. Recall that a permutation of length k is an array of size k such that each integer from 1 to k occurs exactly once. Prefix maximums are the elements that are the maximum on the prefix ending at that element. More formally, the element a_i is a prefix maximum if a_i > a_j for every j < i. Similarly, suffix maximums are defined; the element a_i is a suffix maximum if a_i > a_j for every j > i. You need to output the number of different permutations that could have been guessed. As this number can be very large, output the answer modulo $10^9 + 7$.",
"input": "Input\nEach test consists of several test cases. The first line contains a single integer t ($1 \le t \le 10^4$) — the number of test cases. Then follows the description of the test cases.\nThe first line of each test case contains three integers n, m_1 and m_2 ($1 \le m_1, m_2 \le n \le 2 \cdot 10^5$) — the length of the permutation, the number of prefix maximums, and the number of suffix maximums, respectively.\nThe second line of each test case contains m_1 integers $p_1 < p_2 < \ldots <$ p_{m_1} ($1 \le p_i \le n$) — the indices of the prefix maximums in increasing order.\nThe third line of each test case contains m_2 integers $s_1 < s_2 < \ldots <$ s_{m_2} ($1 \le s_i \le n$) — the indices of the suffix maximums in increasing order.\nIt is guaranteed that the sum of n over all test cases does not exceed $2 \cdot 10^5$.",
"output": " Output\nFor each test case, output a single integer on a separate line — the number of suitable permutations modulo $10^9 + 7$.",
"sample-test": {"input": ["6\n1 1 1\n1\n1\n4 2 3\n1 2\n2 3 4\n3 3 1\n1 2 3\n3\n5 3 4\n1 2 3\n2 3 4 5\n20 5 4\n1 2 3 4 12\n12 13 18 20\n6 2 3\n1 3\n3 4 6"],"output": ["1\n3\n1\n0\n317580808\n10"]},
"note": "Note\nThe following permutations are suitable for the second test case:\n[1, 4, 3, 2]\n[2, 4, 3, 1]\n[3, 4, 2, 1]\n\nThe following permutations are suitable for the sixth test case:\n[2, 1, 6, 5, 3, 4]\n[3, 1, 6, 5, 2, 4]\n[3, 2, 6, 5, 1, 4]\n[4, 1, 6, 5, 2, 3]\n[4, 2, 6, 5, 1, 3]\n[4, 3, 6, 5, 1, 2]\n[5, 1, 6, 4, 2, 3]\n[5, 2, 6, 4, 1, 3]\n[5, 3, 6, 4, 1, 2]\n[5, 4, 6, 3, 1, 2]",
"rating": 2200,
"tags": ["combinatorics","dp","math","number theory"],

Figure 8: An example of original coding problem we obtained in stage I.To make the content more clear, we remove html denotions that contained in original content. The original problem is 1946E, which is used for illustration in Figure 3

.

ple that fit URL anchor identification technique in shown in Figure 9a.

However, there are still little parts of early editorial blogs didn't include URL hyperlinks, so in those cases we fall back on regular-expression–based text matching wherever possible. Based on the formats we observed, we designed two main matching strategies:

- **Difficulty-label matching.** A number of blogs publish all of a round's problems on one page and mark them with labels like "Div2" or "Div1" (since most rounds contain two Div2-level problems and several Div1-level ones). To handle this, we use our problem-ID reference table to identify all sub-IDs belonging to the same numeric contest but with different difficulty levels, tag them accordingly in the table, and then, during the anchor-matching process, if the scraper detects a difficulty label it will also try to match anchors based on that label. An example is shown in Figure 9b.

- **Problem-label matching.** Beyond difficulty tags, many blogs use the literal "Problem A", "Problem B", etc. format. We include a specific regex pattern to detect those "Problem+sub-ID" labels and assign each section to the correct sub-problem.

After crawling the editorial for each problem, we applied a series of processing steps to ensure quality.Firstly, we removed any editorials that were too short or empty—these problems were excluded from the dataset. To make it easier for an LLM to understand and process them, we then split each editorial into two parts: the code solution and the textual explanation, so that the model can consult the code first and then the accompanying text.The processed result is shown in Figure 10.

A.3 Detail of Solution Code Generation

Although we've already scraped the official solution for each problem, an LLM-based post-processing step is still required for two main reasons:**(1) Presence of "global code segments".** Some solutions include code that isn't encapsulated in any function. We must split the entire codebase into multiple functions and ensure that the top-level function can fully solve the problem. Since these global segments can't be recognized during standard parsing, we rely on an LLM to reorganize the provided code so that it becomes fully parsable.**(2)Early solutions exist only as text.**Some of the older official solutions consist solely of textual descriptions without any runnable code. We need an LLM to convert those narratives into executable code.The prompt template for code convert is shown in Figure 11 The Codeforces official judging system is used to verify code correctness. We employ an automated submission bot that navigates to the Codeforces submission page[4], fills in all required fields, and submits the solution. The site will be automatically redirects to the results page after submission, from which we scrape the verification outcome.

A.4 Detail of Subproblems Generation

While the parsing process is automatically, a LLM is still needed for generating natural language description for each subproblem.The prompt template is shown in Figure 12 A example of subproblems in shown in Figure 13.

A.5 Detail of Test Cases Generation

Overall, CodeFlowBench's test suite is composed of two parts:

- **Top-level function tests** For the final subproblem—the top-level function (e.g., `main` or `solve`) that handles overall input and output—we use the public test cases provided by the Codeforces platform.

- **Subproblem function tests** For every other subproblem (i.e. functions invoked by higher-level code), we wrap each function call in a helper that redirects `stdin` and `stdout` to an internal buffer and records the resulting I/O. This can generate redundant calls for the same function, so we apply two safeguards to keep the test suite concise: deduplication of identical test cases and limiting the total number of test cases per function.These measures ensure comprehensive coverage without unnecessary duplication or excessive test-case volume.

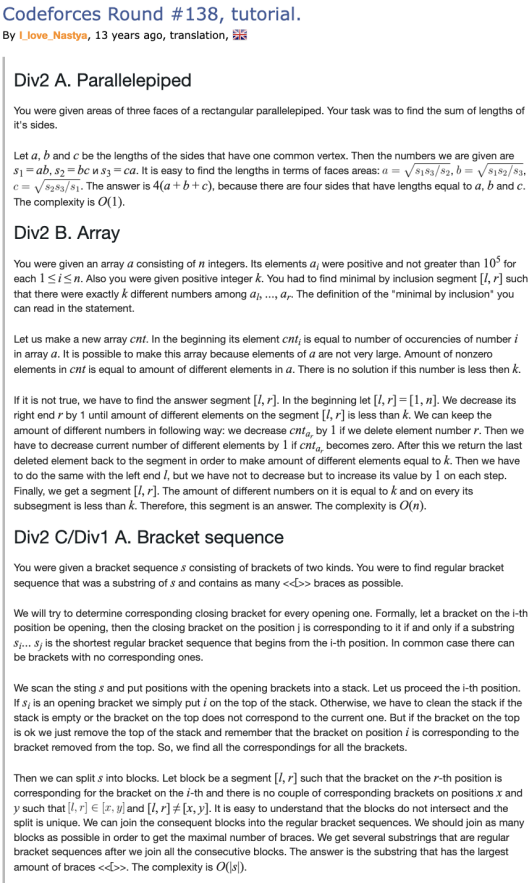
Such method can ensure the test case generation pipeline to be automatic and the case itself to be correct.

A.6 Ablation Study of LLM usage

First, to check its influence on the content and style of Subproblem Descriptions, we compared

[4] https://codeforces.com/problemset/submit

(a)



(b)

Figure 9: Subfigure (a) is an example of using URL anchor identification technique. The anchor here is each subtitle displayed as "2060A-Fibonacciness". The blue font color of such subtitle indicates a URL to original problem page is setted. In practice, the existence of URL is identified by analysing the HTML code of this website.For this kind of website, we identify these subtitles as anchors and scrape the content between each subtitle.Subfigure (b) is an example of using div anchor identification technique.The anchor here is each subtitle displayed as "Div2A. Parallelepiped".These kinds of subtitle occurs in a contest round that contain two div level problems.We use reference table to project the div notation to original problem id and then identify them as corresponding anchor.

"problem-id": "1946E",
"solutions":
[{
"type": "text",
"content": "Editorial First, if p_1 is not equal to 1, or s_{m_2} is not equal to n, or p_{m_1} is not equal to s_1, then the answer is 0 for obvious reasons... Otherwise, we know exactly where the number n is located, at position s_1. Next, we have \binom{n - 1}{s_1 - 1} ways to divide the numbers from 1 to n - 1 into two sets — the numbers that will be in the left part and the numbers that will be in the right part (the left part — all indices < s_1, the right part — all indices > s_1). We solve for the left part, and similarly for the right part. For the left part, the position of the maximum (p_{m_1 - 1}) is again defined, and the maximum itself is also unique in the set of numbers for the left part, so we can again divide the left part into two, with \binom{p_{m_1} - 2}{p_{m_1 - 1} - 1} ways to do so, but we can also arrange the numbers between indices p_{m_1 - 1} and p_{m_1} (non-inclusive) in any order, i.e., (p_{m_1} - p_{m_1 - 1} - 1)! ways. Then we solve similarly for the left set (i.e., for indices less than p_{m_1 - 1}).
},
{
"type": "code",
"content": "#include<bits/stdc++.h>\nusing i64 = long long;\ntemplate<class T> constexpr T power(T a, i64 b) {...}"
}]

Figure 10: The scraped and processed solution we obtained in stage II. The original problem is 1946E

.

Deepseek-V3 against GPT-4o-mini, Gemini-2.0-Flash and o3-mini. As shown in Table 4, the high semantic overlap and consistent stylistic metrics indicate that descriptions are primarily driven by the underlying code logic rather than model-specific habits. The main content of subproblems is not highly influenced by model choice.

Second, to check its influence on the structure of LLM-generated solution code, we re-generated solutions using o3-mini, Claude-3.7-Sonnet, and Gemini-2.5-Pro. The resulting complexity metrics (Table 5) were identical across all models, confirming that the model produces almost identical decomposition results, indicating very minor bias between different LLMs.

Finally, to verify that our subproblems are natural and representative of human problem-solving, we conducted a Human-LLM cross-validation by constructing a creteria including five aspects. As shown in Table 6, both human and model judges consistently rated the subproblems as high-quality, which confirms that our automated pipeline produces professional, human-aligned problem decompositions.

Prompts of Subproblem Description Quality Validation is shown as follows:

Subproblem Description Quality Validation

You are an expert evaluator of programming task subproblems.
You will receive a subproblem (including the statement, solution code, and test cases). Evaluate whether the statement adequately meets each of the following five criteria. For each criterion, assign **1** if it is satisfied, or **0** if it is not. The total possible score is thus 5.

1. Clarity: Is the statement smoothly and understandably expressed, allowing a reader to grasp the task goal quickly?
2. Completeness: Does the description include all key elements needed to accomplish the subtask?
3. Accuracy: Is the description free of ambiguity or logical errors, and does it match the problem requirements and the provided code?
4. Feasibility: Can an engineer unambiguously determine and implement the required functionality—and pass functional tests—based solely on this description?
5. Professionalism: Does it use accurate, domain-appropriate terminology and a style fitting technical norms of coding tasks?

Return your result **only** as a JSON dictionary with these five keys and values of 0 or 1.

Please do not be overly strict. Assign a score of 1 to a criterion if it is even partially satisfied, allowing for minor imperfections. Only give a score of 0 if there are major or fundamental issues. Furthermore, ensure that the overall score for any subproblem is not lower than 2.
Here is the subproblem:content

Only return the JSON dictionary and remove the "'json note, nothing else.

Here's the information for the problem:
"problem-id": "{problem_id}",
"title": "{title}",
"time_limit": "{time_limit}",
"memory_limit": "{memory_limit}",
"problem_description": "{problem_description}",
"input": "{problem_input}",
"output": "{problem_output}",
"sample-test": "{sample_testcases}",
"note": "{note}",

Here's the editorial of this problem:
"solution_text_part": "{text_solution}"
"solution_code_part": "{code_solution}",

Your task is to generate a correct Python code solution that solves the problem, here's the instruction:
1. If the "solution_code_part" is not empty and contain vaild code, you should only focus on that part. If the original code is not in Python, convert it to Python. You can directly check the code and output it as your generated code.
2. Remeber if the "solution_code_part" is not empty and contain vaild code, use the existing code is always the priority, but if the "solution_code_part" is empty, you should analyze the "solution_text_part" field together with the problem details provided above and generate the Python code-solution that implements the solution with the following characteristics:
• There should be a main/solve function that handle the overall logic and I/O.
• Each function should have a single responsibility.
• The main solver function should handle input/output and orchestrate the helper functions.
• Ensure the code is properly indented and formatted. Remember to use 4 spaces for indentation.
• Ensure that you have set proper new line notation before the 'if __name__ == "__main__":' line and "import" statements.
• Avoid nest function in another function, make each function seperated.
• Each line of code should be contained in a function except calling packages.
3. At last, return your generated code in the “code-solution” field of the output JSON, following the format specified in the system instructions.
Example output data:
```json
{{
"problem-id": "2055E",
“code-solution”: “…”
}}
```
Please ensure the code you generate can solve the problem correctly, whicn means:
- The code should be free of syntax errors.
- The code should run correctly on the sample test provided and other egde cases.
- The code should have optimal time and memory complexity to solve the problem without surpassing the time and memory limits.
- Each line of code should be contained in a function except calling packages.
You only need to return the json with the "code-solution" field filled following the "```json" mark in the next line.
```json

Figure 11: The prompt template used for code convertion in stage III.The whole content of the example output data is not shown for length limitation.
```

Here is the problem information and analyzed functions:
Title: {title}
Time Limit: {time_limit}
Memory Limit: {memory_limit}
Problem Description: {problem_description}
Input: {problem_input}
Output: {problem_output}
Note: {note}
Extracted Functions:
{json.dumps(subproblems, indent=4)}
Text Solution:{solution_text_part}
Missing Class:{missing_class}
Your task is to:
1. Based on the function that "Extracted Functions" field contains, for each Class and Function , use the problem information provided to write a subproblem description that includes:
- Name of the function/class.
- The purpose of the function/class.
- How it contributes to solving the overall problem.
- Any dependencies it has on other functions. Set a reminder of functions that need to be called.
2. Return the results in the following JSON format. Use the subproblem statement you write to fill in the "statement" field and the information of each function to fill the "depth" and "dependencies" fields.
```json
{{
"problem-id": "{problem_id}",
"subproblems": [
{{
"name": "...",
"statement": "...",
"solution":"...",
"depth":"...",
"dependencies":[...]
}},
]
}}
```
Reminder:
- Please follow the system instruction and ensure that each function is described accurately and comprehensively. If a function has no dependencies, you can just return the main or top-level function description.
- For the solution code of each subproblem, please return the whole complete function definition, including the function signature, parameters, and body.
- You need to output the subproblems following the dependency order, starting from the functions that have no dependencies and let the function which depends on other functions be described later.
- Don't forget to include both the whole Class and the function inside the Class as solution in the solution code as a subproblem, otherwise there will be a gap between the funciton in the class and outside the class.
You only need to return the json with the "subproblems" list filled following the "```json" mark in the next line.
```json

Figure 12: The prompt template used for generating natural language description for each subproblem in stage III.
```

```
”problem-id”: "1946E",
"subproblems":
[
    {
    "name": "comb",
    "statement": "Compute the binomial coefficient C(n, k) modulo 10^9 + 7. This function is essential for calculating
    the number of ways to choose k elements from a set of n elements, which is a key part of determining the number of
    valid permutations in the problem. The function uses precomputed factorials and inverse factorials to efficiently
    compute the result.",
    "solution": "MOD = 10 ** 9 + 7\nfact = [1] * max_fact\ninv_fact = [1] * max_fact\ndef comb(n, k):\n if n < 0 or k
    < 0 or n < k:\n return 0\n return fact[n] * inv_fact[k] % MOD * inv_fact[n - k] % MOD",
    "depth": 1,
    "dependencies": [],
    },
    {
    "name": "solve",
    "statement": "Read input, process each test case, and compute the number of valid permutations. The function first
    checks if the given prefix and suffix maximums are valid. If not, it outputs 0. Otherwise, it calculates the number of
    valid permutations by dividing the problem into left and right parts, using the `comb` function to compute binomial
    coefficients and factorials to account for the arrangement of elements between indices. The result is computed
    modulo 10^9 + 7 and printed for each test case.",
    "solution": "MOD = 10 ** 9 + 7\nfact = [1] * max_fact\ndef solve():\n import sys\n input =
    sys.stdin.read().split()\n ptr = 0\n t = int(input[ptr])\n ptr += 1\n for _ in range(t):\n n = int(input[ptr])\n m1 =
    int(input[ptr + 1])\n m2 = int(input[ptr + 2])\n ptr += 3\n p = list(map(int, input[ptr:ptr + m1]))\n ptr += m1\n s =
    list(map(int, input[ptr:ptr + m2]))\n ptr += m2\n if p[0] != 1 or s[-1] != n or p[-1] != s[0]:\n print(0)\n continue\n s0
    = s[0]\n res = comb(n - 1, s0 - 1)\n for i in range(m1 - 2, -1, -1):\n next_p = p[i + 1]\n current_p = p[i]\n k = next_p -
    current_p - 1\n c = comb(next_p - 2, k)\n f = fact[k]\n res = res * c % MOD\n res = res * f % MOD\n for i in range(1,
    m2):\n prev_s = s[i - 1]\n current_s = s[i]\n total = n - prev_s - 1\n k = current_s - prev_s - 1\n c = comb(total, k)\n f
    = fact[k]\n res = res * c % MOD\n res = res * f % MOD\n print(res)",
    "depth": 0,
    "dependencies":["comb"],
    }
]
```


Figure 13: An example of subproblems we obtained in stage IV.The solution code of 1946E contains a `comb` function which serve as the basic tool and is reused in `solve` function to address the whole problem.It's obvious that in its AST, the `comb` function is the leave node in depth 1 and the `solve` function is the root node in depth 0.

Table 4: Robustness of Subproblem Descriptions (Ref: Deepseek-V3). **Left**: Semantic Consistency; **Right**: Stylistic Similarity.

Model	Semantic		Stylistic		
	ROUGE-1	BERT	Len. Ratio	Marker Diff.	Error Diff.
GPT-4o-mini	0.582	0.725	0.975	0.026	0.367
Gemini-2.0-Flash	0.600	0.734	1.141	0.019	0.728
o3-mini	0.554	0.692	1.181	0.124	1.350

Table 5: Structural Complexity Metrics Across Models

Model	Avg. Turns	Avg. Depth
o3-mini	2.08	1.94
Claude-3.7-Sonnet	2.08	1.94
Gemini-2.5-Pro	2.08	1.94

A.7 Compute Cost Estimation

The cost of using LLM in our data curation pipeline is estimated as below:

- Generating Solution Code: On average, each problem requires approximately 3K input tokens and 5K output tokens (including reasoning). The official Deepseek-R1 API is currently 4 CNY per 1M input tokens and 16 CNY per 1M output tokens. Therefore, the total cost for 5K+ problems is approximately 450 CNY.
- Generating Subproblem Description: On average, each problem requires approximately 3K input tokens and 1K output tokens. The official Deepseek-V3 API is currently 2 CNY per 1M input tokens and 8 CNY per 1M output tokens. Therefore, the total cost for 5K+ problems is approximately 70 CNY.

The cost of processing the initial batch of 5K+ problems is manageable and also a one-time effort that does not need to be repeated. Future updates to the benchmark will involve only newly published problems and will therefore be much smaller in scale and less resource-intensive.

B CodeFlowBench-Repo Dataset

In this section, we provide a detailed overview of the *CodeFlowBench-Repo* dataset.

B.1 DomainEval

Overview of DomainEval DomainEval is an auto-constructed benchmark specifically designed to evaluate LLMs' coding capabilities across diverse real-world domains. Unlike general-purpose benchmarks that focus on common algorithmic tasks, DomainEval targets six distinct domains: *Computation, Network, Basic, System, Visualization, and Cryptography*. The dataset is curated from representative open-source GitHub repositories that possess high community recognition (typically over 100 stars) to ensure code quality.

Table 6: Human Evaluation of Subproblem Quality. Metrics include Clarity (Cla.), Completeness (Com.), Accuracy (Acc.), Feasibility (Fea.), and Professionalism (Pro.).

Source	Avg. Rating	Cla.	Com.	Acc.	Fea.	Pro.
o3-mini	**4.222**	0.981	0.604	0.955	0.797	0.986
Claude-3.7	4.038	0.889	0.745	0.745	0.668	0.990
Human	4.202	0.981	0.659	0.904	0.726	0.933

Structurally, each subject in DomainEval consists of three essential components: a natural language instruction, a reference solution (target function), and a suite of valid test cases extracted via a test-method matching strategy. Crucially, to ensure executability, DomainEval preserves the necessary context required to run the target code. This design ensures that the tasks are not isolated snippets but are deeply embedded in the dependency structures of authentic software projects.

Rationale for Source Selection. The selection of DomainEval as the foundation for *CodeFlowBench-Domain* is a deliberate design choice driven by two key factors:

1. Authenticity across Specialized Domains. First, DomainEval provides a rigorous source of ecologically valid code samples. By extracting problems from mature, production-level repositories, it captures the authentic coding patterns that professional developers actually employ. Furthermore, its explicit coverage of specialized domains like *Cryptography* and *System* areas where general LLMs often struggle allows our benchmark to assess Dependency Awareness in contexts that require precise API knowledge, effectively complementing the algorithmic logic focus of standard competition datasets.

2. Alignment with Constructive Codeflow Paradigm.

While many repository-level benchmarks focus on *maintenance* tasks such as issue resolution or debugging within existing codebases, CodeFlowBench aims to evaluate the constructive aspect of software engineering, namely to build complex functionality from the ground up. DomainEval focuses on function generation within a rich context.

This format is the ideal input for our pipeline, allowing us to decompose a monolithic, dependency-heavy function into a step-by-step, iterative construction process. By leveraging DomainEval, we effectively demonstrate that the iterative codeflow paradigm is adaptable to building functional modules in authentic software libraries, validating our pipeline's effectiveness beyond algorithmic puzzles.

B.2 Dataset Construction

The construction of the CodeFlowBench-Repo subset adapts the Subproblem Generation Phase from the CodeFlowBench-Comp pipeline. A example of problems in DomainEval is shown in Fig. 14

Since DomainEval already provides high-quality reference functions as ground truth, we omit the verification step and proceed directly to dependency analysis. We remove problems which only include single-turn function from the dataset and for the remaining, we parse the dependency relationship of the reference code to identify internal function calls and decompose them into iterative subproblems. Similarly, we then call LLM to generate problem description for subproblems. Finally, we use test cases of the top-level function to generate test cases of subproblems. An example is shown in Fig. 15.

To ensure robust evaluation, we parse the import statements from the original test files and utilize Docker containers to encapsulate the complex package dependencies required by these real-world repositories. This ensures a compatible and reproducible execution environment for all generated subproblems.

B.3 Dataset Statistics

Since our pipeline explicitly looks for internal function reuse to create multi-turn steps, the API-centric nature of the Computation domain resulted in empty dependency trees. Consequently, our selection was restricted to the remaining 749 problems from the other five domains. After further filtering, only 70 high-quality multi-turn problems survived the pipeline. To ensure a balanced evaluation, we augmented this set by add 12 single-turn problems each domain and curate the final CodeFlowBench-Repo.

As illustrated in Fig. 16, the dataset features a balanced distribution across five distinct domains: System (28.7%) and Cryptography (23.3%) constitute the majority, followed by Network (21.7%), Basic utilities (17.8%), and Visualization (8.5%). This distribution ensures the benchmark covers a wide spectrum of software engineering scenarios, from low-level OS interactions to high-level data plotting.

Structurally, the dataset presents significant complexity beyond atomic code generation. The distribution of overall turns and overall depths is shown in Fig. 16. The distribution of overall turns reveals that 56.6% of the problems require multi-turn iterative generation, forcing models to maintain reasoning continuity over long horizons. Furthermore, the dependency depth distribution confirms the intricate nature of these tasks: while 45.7% of problems act as foundational nodes (Depth 1), the remaining 54.3% involve deeper dependency chains (Depth 2–4), requiring the model to correctly implement and call prerequisite functions within a hierarchical context.

C Mathematical Expression of $PD@k$& $APD@k$

For a given problem, we define $PD@k$ as the expected maximum pass depth over k independent trials of the model. Directly using only those k results leads to high variance, just as with the $pass@k$ metric. By analogy to the unbiased estimator for $pass@k$, which leverages n trials ($n > k$) to reduce variance, we derive a similar estimator for $PD@k$.

Let the pass depths from n trials be:

$$\{d_1, d_2, \ldots, d_n\},$$

and let

$$d_{(1)} \leq d_{(2)} \leq \cdots \leq d_{(n)}$$

denote these depths in ascending order. Then an unbiased estimator for $PD@k$ is

$$PD@k \;=\; \sum_{j=k}^{n} d_{(j)} \,\frac{\binom{j-1}{k-1}}{\binom{n}{k}}. \tag{1}$$

To see how this arises, consider sampling a random subset of size k from the n depths. Let $M = \max\{\, d_{i_1}, d_{i_2}, \ldots, d_{i_k}\}$ be the maximum depth in that subset. An unbiased estimator of M is $\mathbb{E}[M]$. By construct $\mathbb{E}[M] = PD@k$,this estimator is *unbiased* since its expected value exactly equals the true expected maximum depth. Consider all possible value of M, we have:

$$\mathbb{E}[M] = \sum_{j=k}^{n} d_{(j)}\, P\big(M = d_{(j)}\big). \tag{2}$$

```
“method_name”: ”unparse",
“full_method_name”: ”unparse",
“ method_path”: ” "../srcdata/Network/mitmproxy/mitmproxy/net/http/url.py ",
“full_method_name”: ”unparse",
“method_code”:"from __future__ import annotations\nimport re\nimport urllib.parse\nfrom collections.abc import
Sequence\nfrom typing import AnyStr\ndef default_port(scheme: AnyStr) ->(int | None):\n return {'http': 80,
b'http': 80, 'https': 443, b'https': 443}.get(scheme,\n None)\ndef hostport(scheme: AnyStr, host: AnyStr, port:
int) ->AnyStr:\n \"\"\"\n Returns the host component, with a port specification if needed.\n \"\"\"\n if
default_port(scheme) == port:\n return host\n elif isinstance(host, bytes):\n return b'%s:%d' % (host, port)\n
else:\n return '%s:%d' % (host, port)\ndef unparse(scheme: str, host: str, port: int, path: str='') ->str:\n
\"\"\"\n Returns a URL string, constructed from the specified components.\n Args:\n All args must be str.\n
\"\"\"\n if path == '*':\n path = ''\n authority = hostport(scheme, host, port)\n return
f'{scheme}://{authority}{path}’”
“test_code_list”:": [{
    "test_code": "from typing import AnyStr\nimport pytest\ndef test_unparse():\n assert unparse('http', 'foo.com', 99, '')
== 'http://foo.com:99'\n assert unparse('http', 'foo.com', 80, '/bar') == 'http://foo.com/bar'\n assert unparse('https', 'foo.com',
80, '') == 'https://foo.com:80'\n assert unparse('https', 'foo.com', 443, '') == 'https://foo.com'\n assert unparse('https',
'foo.com', 443, '*') == 'https://foo.com'\n\ntest_unparse()\n",
    "code_start": "",
    "test_path": "../srcdata/Network/mitmproxy/test/mitmproxy/net/http/test_url.py”
    }]
“instruction”: "Functionality: Constructs a URL string from given components.\nInputs: \n- scheme: A string
representing the URL scheme (e.g., \"http\", \"https\").\n- host: A string representing the host component of the URL.\n-
port: An integer representing the port number.\n- path: An optional string representing the path component of the URL
(default is an empty string).\n\nOutputs:\n- A string representing the constructed URL. The function ensures that the port
is included in the URL only if it does not match the default port for the given scheme. If the path is set to \"*\", it is
treated as an empty string.",
“method_code_task”: "from __future__ import annotations\nimport re\nimport urllib.parse\nfrom collections.abc
import Sequence\nfrom typing import AnyStr\n\n\ndef default_port(scheme: AnyStr) ->(int | None):\n return {'http':
80, b'http': 80, 'https': 443, b'https': 443}.get(scheme,\n None)\n\n\ndef hostport(scheme: AnyStr, host: AnyStr,
port: int) ->AnyStr:\n \"\"\"\n Returns the host component, with a port specification if needed.\n \"\"\"\n if
default_port(scheme) == port:\n return host\n elif isinstance(host, bytes):\n return b'%s:%d' % (host, port)\n
else:\n return '%s:%d' % (host, port)\n\n\ndef unparse(scheme: str, host: str, port: int, path: str='') ->str:
[MASK]\n"
```


Figure 14: An example problem in DomainEval. We use the "method_code" attribute to generate subproblem solution code and use test case in "test_code" to generate subproblems’ test cases.

```
”problem-id”: ” Network__mitmproxy__unparse",
”problem-description”: "Functionality: Constructs a URL string from given components.\nInputs: \n- scheme: A string representing the URL scheme (e.g., \"http\", \"https\").\n- host: A string representing the host component of the URL.\n- port: An integer representing the port number.\n- path: An optional string representing the path component of the URL (default is an empty string).\n\nOutputs:\n- A string representing the constructed URL. The function ensures that the port is included in the URL only if it does not match the default port for the given scheme. If the path is set to \"*\", it is treated as an empty string.",
"domain": "Network",
"type": "multi",
"source_file": "mitmproxy",
"solution": "from __future__ import annotations\nimport re\nimport urllib.parse\nfrom collections.abc import Sequence\nfrom typing import AnyStr\ndef default_port(scheme: AnyStr) ->(int | None):\n return {'http': 80, b'http': 80, 'https': 443, b'https': 443}.get(scheme,\n None)\ndef hostport(scheme: AnyStr, host: AnyStr, port: int) ->AnyStr:\n \"\"\"\n Returns the host component, with a port specification if needed.\n \"\"\"\n if default_port(scheme) == port:\n return host\n elif isinstance(host, bytes):\n return b'%s:%d' % (host, port)\n else:\n return '%s:%d' % (host, port)\ndef unparse(scheme: str, host: str, port: int, path: str='') ->str:\n \"\"\"\n Returns a URL string, constructed from the specified components.\n Args:\n All args must be str.\n \"\"\"\n if path == '*':\n path = ''\n authority = hostport(scheme, host, port)\n return f'{scheme}://{authority}{path}'",
"test_code": [{
    "test_code": "from typing import AnyStr\nimport pytest\ndef test_unparse():\n assert unparse('http', 'foo.com', 99, '') == 'http://foo.com:99'\n assert unparse('http', 'foo.com', 80, '/bar') == 'http://foo.com/bar'\n assert unparse('https', 'foo.com', 80, '') == 'https://foo.com:80'\n assert unparse('https', 'foo.com', 443, '') == 'https://foo.com'\n assert unparse('https', 'foo.com', 443, '*') == 'https://foo.com'\n\ntest_unparse()\n",
    "code_start": "",
    "test_path": "../srcdata/Network/mitmproxy/test/mitmproxy/net/http/test_url.py"
}],
"subproblems": [
    {
    "name": "default_port",
    "statement": "The default_port function determines the default port number for a given URL scheme. It...",
    “solution”: “…",
    "depth": 0,
    "dependencies": [],
    "test_code": [
    {"input": "(('http',), {})","output": "80"},
    {"input": "(('https',), {})","output": "443"}
    ]},
    {
    "name": "hostport",
    "statement": "The hostport function constructs the host component of the URL, optionally including the port number if it differs from the default port for the specified scheme. It uses the default_port function to …",
    “solution”: ”…",
    "depth": 1,
    "dependencies": ["default_port"],
    "test_code": [
    {"input": "(('http', 'foo.com', 99), {})","output": "'foo.com:99'"},
    {"input": "(('http', 'foo.com', 80), {})","output": "'foo.com'"},
    {"input": "(('https', 'foo.com', 80), {})","output": "'foo.com:80'"},
    {"input": "(('https', 'foo.com', 443), {})","output": "'foo.com'"}
    ]},
    {
    "name": "unparse",
    “statement”: “The unparse function constructs a complete URL string from its components: scheme, host, port, and path. It…",
    "solution": ”…'",
    "depth": 2,
    "dependencies": ["hostport"],
    "test_code": [
    {"input": "(('http', 'foo.com', 99, ''), {})","output": "'http://foo.com:99'"},
    {"input": "(('http', 'foo.com', 80, '/bar'), {})","output": "'http://foo.com/bar'"},
    {"input": "(('https', 'foo.com', 80, ''), {})","output": "'https://foo.com:80'"},
    {"input": "(('https', 'foo.com', 443, ''), {})","output": "'https://foo.com'"},
    {"input": "(('https', 'foo.com', 443, '*'), {})","output": "'https://foo.com'"}]
}],
"overall-depth": 3,
"overall-turns": 3
},
```


Figure 15: An example problem in CodeFlowBench-Repo, which has a shared form of problems in CodeFlowBench-Repo.

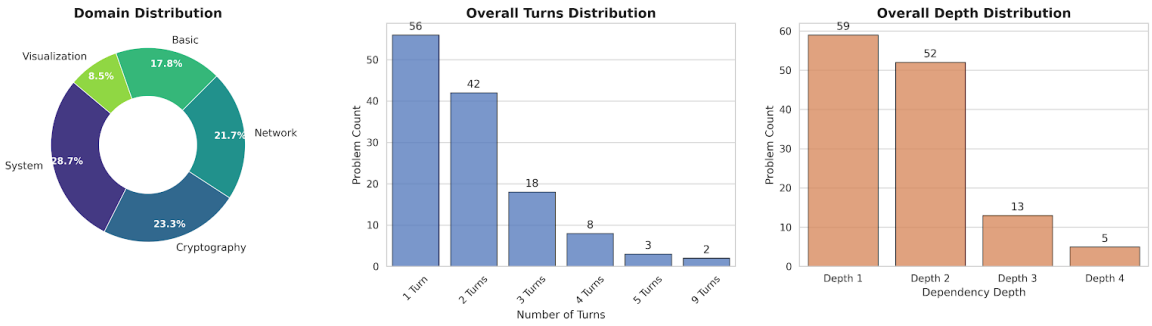


Figure 16: Statistics of CodeFlowBench-Repo. Left: The diversity of domains, with a focus on System and Cryptography tasks. Right: Distributions of interaction turns and dependency depths, highlighting the structural complexity.

The probability that $M = d_{(j)}$ equals the probability of choosing $d_{(j)}$ along with $k-1$ depths from the first $j-1$ smaller values:

$$P\big(M = d_{(j)}\big) = \frac{\binom{j-1}{k-1}}{\binom{n}{k}}. \tag{3}$$

Combining (3) and (4) immediately recovers (1).

Finally, we define

$$APD@k = \frac{1}{|\mathcal{P}|} \sum_{p \in \mathcal{P}} PD@k(p) \tag{4}$$

the average $PD@k$ over the set $\mathcal{P}$ of all problems.

D Experiment

D.1 Experiment Setting

Due to the substantial size of the total question pool ($N = 5258$), we implemented stratified sampling with proportional allocation across overall-depth categories to select 1,000 test samples, as detailed in Table 7. To validate sampling quality, we conducted a χ^2 test comparing the overall-turn distributions between the population and sampled data (Table 8). The statistical analysis yielded a p-value of 0.1246, indicating no significant difference ($\alpha = 0.05$) in distribution patterns. This confirms the representativeness of our sampling strategy and ensures the validity of subsequent analytical outcomes.

To ensure the reproducibility of our results, we detail the inference hyperparameters in Table 9.

D.2 Instruction Templates

In the instruction template for multi-turn testing, we implemented a set of basic heuristics to adapt to different problem types. For example, if a question has no dependencies, we omit the "## Dependency information" section. If no prior code is provided—which is common when the overall depth is 1—we exclude the "## Pre-implemented functions" section. Furthermore, if it is the final turn of the code, we append the following snippet:

```
import sys
def {name}():
    input = sys.stdin.read().split()
```

Similarly, if the problem does not have any dependencies, we will also omit the section related to {dependencies}.

Multi-turn Test

You are a Programming Expert. You always provide correct and reliable code solutions. You will be provided with the background of the whole problem, a programming problem and may also some pre-implemented functions.If pre-implemented functions provided, you need to call the pre-implemented functions and write a new function to solve the problem.
Background of the whole problem:
{problem_description}
Problem Description: You need to complete name function.
{statement}
Dependency information:
To solve the problem, you need to utilize the ## Pre-implemented functions {dependencies} provided.
Pre-implemented functions:
{history}
Guidelines:
- Ensure the function is executable and meets the requirement.
- Handle ## Dependency information correctly.
- Provide clear and concise comments to explain key parts of the code.
Return your response by filling the function body following the function signature provided. Just generate the function and don't output any examples.

Table 7: Comparison of Overall-Depth Proportions Between the Population and the Sample

Overall-Depth	# Population	Pop. Proportion (%)	# Sample	Sam. Proportion (%)
1	1,488	28.3	283	28.3
2	2,751	52.3	523	52.3
3	870	16.5	165	16.5
4	125	2.4	24	2.4
≥5	24	0.5	5	0.5
Total	**5,258**	**100.0**	**1,000**	**100.0**

Table 8: Comparison of Overall-Turn Proportions Between the Population and the Sample

Overall-Turn	# Population	Pop. Proportion (%)	# Sample	Sam. Proportion (%)
1	1,488	28.3	283	28.3
2	2,158	41.0	437	43.7
3	990	18.8	188	18.8
4	402	7.6	58	5.8
5	149	2.8	21	2.1
6	46	0.9	6	0.6
≥7	25	0.5	7	0.7
Total	**5,258**	**100.0**	**1,000**	**100.0**

Single-turn Test

You are a Programming Expert. You always provide correct and reliable code solutions. You are required to solve a problem which consists of multiple subproblems, each with its own requirements. You will be provided with the background of the problem and description of all subproblems. You need to generate the complete implementations for all subproblems in a single response. The response for the final subproblem will be tested using stdin and stdout. Ensure the corresponding code meet this requirement.
Background of the whole problem:
{problem_description}
Problem Description:
{combined_subproblem_description}
Subproblem {name}
Description:
You need to complete {name} function.
{statement}
To solve the problem, you need to utilize your pre-implemented function {dependencies}.
Guidelines: - Ensure that all functions are executable and meet their respective requirements.
- For each subproblem, correctly handle any dependency information.
- Provide clear and concise comments explaining the key parts of the code.
- For the last subproblem name, please use 'import sys\ndef {name}():\n input = sys.stdin.read().split()\n" as the beginning.
Return your response by generating all functions in a single code block.

D.3 Statistic Rigor

To ensure the statistical rigor of our empirical findings and verify that the observed performance gaps are not merely artifacts of data sampling variance, we re-evaluated our main results by constructing 95% confidence intervals.

Given the nature of pass rates (Pass@k) and Average Pass Depth (APD@k), which may not strictly follow a normal distribution, we employed non-parametric bootstrapping. For each dataset and model combination, we generated $N = 1000$ bootstrap resamples (with replacement) from the original test sets. We calculated the metrics for each resample to construct a bootstrap distribution. The standard error (SE) was derived from the standard deviation of this distribution, which was then used to approximate the 95% confidence interval.

Table 10 presents the performance of representative models across different tiers, augmented with the computed bootstrap standard errors.

E Ablation Study

E.1 Empirical Validation of Software Quality

While Pass@k metrics effectively capture the functional correctness of generated code, they fall short in assessing its maintainability and alignment with

Table 9: Inference Hyperparameters for Evaluation

Hyperparameter	API	Local	Description
Temperature	0.6	0.6	Balances creative reasoning with logical consistency.
Top-p	1.0	1.0	No nucleus sampling truncation applied.
Max Tokens	5,000	10,000	Maximum tokens generated per subproblem turn.
Frequency Penalty	0.0	0.0	Prevents repetitive token generation.

Table 10: Main Results with Bootstrap Confidence Intervals ($\pm$ Standard Error, 95% CI via 1000 resamples). Pass@1 values are reported as percentages.

Model	Multi-Turn (Pass@1 %)	Single-Turn (Pass@1 %)	Overall APD@1 ($\pm$ SE) across Depths			
			Depth 1	Depth 2	Depth 3	Depth 4
CodeFlowBench-Comp						
GPT-5	26.5 ± 1.4	37.6 ± 1.7	0.185 ± 0.023	0.866 ± 0.039	1.306 ± 0.092	1.542 ± 0.284
Gemini-3-flash-thinking	48.4 ± 1.6	65.5 ± 1.5	0.445 ± 0.030	1.303 ± 0.040	1.625 ± 0.102	1.625 ± 0.321
Claude-4.5-Sonnet	31.6 ± 1.5	47.9 ± 1.6	0.222 ± 0.024	1.008 ± 0.040	1.377 ± 0.088	1.521 ± 0.248
Qwen3-Coder-30B-A3B	11.6 ± 1.0	26.0 ± 1.3	0.100 ± 0.018	0.458 ± 0.032	0.681 ± 0.070	0.917 ± 0.194
CodeFlowBench-Repo						
GPT-5	34.7 ± 4.2	48.8 ± 4.3	0.483 ± 0.066	1.020 ± 0.110	1.385 ± 0.340	3.200 ± 0.429
Gemini-3-flash-thinking	33.9 ± 4.2	56.7 ± 4.4	0.500 ± 0.066	1.020 ± 0.113	1.308 ± 0.338	3.400 ± 0.358
Claude-4.5-Sonnet	33.1 ± 4.2	56.7 ± 4.3	0.448 ± 0.064	1.039 ± 0.111	1.615 ± 0.323	3.200 ± 0.436
Qwen3-Coder-30B-A3B	23.6 ± 3.8	42.5 ± 4.2	0.414 ± 0.064	0.471 ± 0.100	1.385 ± 0.376	0.600 ± 0.282

software engineering best practices. To comprehensively evaluate whether the codeflow paradigm intrinsically yields higher code quality than monolithic generation, we conducted an empirical ablation study focusing on structural and maintainability metrics.We utilized standard static analysis tools (e.g., `radon` for Python) to extract the following metrics from the generated solutions:

- **Average Function Count (FC):** A proxy for modularity. A higher count in solving the same problem indicates a stronger inclination toward problem decomposition.
- **Average Function Length (SLOC):** Source Lines of Code per function. Shorter functions generally indicate better encapsulation and strict adherence to the Single Responsibility Principle (SRP).
- **Comment Density (CD):** The ratio of comment lines to total lines, reflecting the code's self-documentation capabilities.
- **Cyclomatic Complexity (CC):** A quantitative measure of the number of linearly independent paths through a program's source code. Lower CC values imply that the code is easier to test, debug, and maintain.

We sampled successful completions for problems requiring $T > 1$ turns across three leading models under both multi-turn and single-turn settings. Table 11 demonstrates the result. Models operating in the codeflow paradigm generated more functions per solution with significantly higher comment density, effectively breaking down complex logic into manageable units. Particularly for Gemini-3 and Claude-4.5, the multi-turn setting led to a notable reduction in both SLOC and Cyclomatic Complexity per function. This confirms that guiding models through iterative subproblems actively prevents the generation of convoluted, monolithic code.

Beyond static metrics, the fundamental advantage of the multi-turn CodeFlow paradigm lies in its workflow alignment. Monolithic single-turn generation creates a "black-box" scenario where errors in logic or integration are notoriously difficult to trace. In contrast, our paradigm enables developers to unit-test atomic modules at each turn. By isolating functionality, errors are constrained to the current working module, significantly reducing the cognitive load required for debugging and mirroring real-world Agile development practices.

Table 11: Maintainability and Complexity Metrics across Models (Multi-turn vs. Single-turn)

Model	Setting	Avg. FC (↑)	Avg. SLOC (↓)	CD (↑)	CC (↓)
Gemini-3	Multi-Turn	3.53	15.14	37.53%	5.11
	Single-Turn	2.75	16.12	33.50%	5.21
Claude-4.5	Multi-Turn	2.98	14.41	33.04%	4.54
	Single-Turn	2.28	20.06	23.56%	6.21
GPT-5	Multi-Turn	2.93	19.35	31.43%	6.62
	Single-Turn	2.21	16.17	23.22%	5.29

E.2 Analysis of Data Contamination Risks

Given that CodeFlowBench sources data from public platforms like Codeforces and GitHub, a natural concern arises regarding potential data contamination. To ensure that our benchmark evaluates true reasoning capabilities rather than the models' memorization of their training corpora, we address this issue through both structural methodology and empirical validation.

Structural Barrier Against Retrieval We emphasize that the CodeFlow task extends far beyond simple code retrieval. Our AST-based pipeline transforms monolithic solutions into constrained, dependency-aware sequences. To succeed, models must demonstrate deep contextual reasoning to adapt the overarching logic into specific subproblem signatures and strict modular boundaries. This structural transformation acts as a natural barrier; simply recalling the original training data or a monolithic script is insufficient to pass the multi-turn tests.

Chronological Contamination Check To provide empirical evidence, we conducted a chronological contamination check on CodeFlowBench-Comp. We evaluated model performance across three distinct chronological slices sampled from the most recent 1,000 problems. To ensure a fair and unbiased comparison, these slices were strictly matched in terms of average difficulty ratings, interaction turns, and dependency depth. Table 12 presents the Pass@1 and APD@1 scores for Gemini-3 and Claude-4.5 across these time periods. As shown, there is no significant or consistent performance degradation on newer problems (Slice 3) compared to older ones (Slice 1). In fact, Claude-4.5 even shows a slight performance increase on newer problems. This lack of negative chronological correlation strongly suggests that the models' performance is driven by their intrinsic reasoning and logical capabilities, rather than the memorization of specific problem instances.

E.3 Analysis of Evaluation Protocol: Fail-Stop vs. Non-Stop

A potential concern regarding our primary evaluation framework is whether the strict "fail-stop" mechanism overly penalizes models for trivial early-stage errors, thereby obscuring their capacity to solve complex logic in subsequent turns. To rigorously isolate the models' logical capabilities at each specific step without the compounding effect of earlier failures, we conducted an ablation study using a "non-stop" (Teacher-Forcing) evaluation variant.

Experimental Setup We sampled 100 problems and evaluated three leading models. In the non-stop mode, if a model fails a specific turn, we inject the ground-truth implementation for that subproblem into the context. This intervention provides the model with a "clean slate" to tackle the subsequent dependencies. To quantify this, we introduce the *Absolute Solved Turns (AST)* metric. Unlike standard pass metrics, AST calculates the absolute number of subproblems a model solves correctly per problem, given that any failed preceding turns are seamlessly replaced with ground-truth implementations.

Results Table 13 contrasts the AST scores under the standard fail-stop setting versus the non-stop setting. While the non-stop mode predictably yields slightly higher scores, the overall performance gains (Δ) are remarkably marginal . This minimal variance provides a crucial insight: a model that fails an early, low-complexity helper function genuinely struggles with the underlying logic and typically lacks the reasoning depth required for the heavier dependencies in subsequent

Table 12: Model Performance Across Chronological Slices (Controlled for Difficulty, Depth, and Turn)

Model	Chronological Slice	ID Range	Pass@1	APD@1
Gemini-3	Slice 1 (Oldest)	1609 - 1826	0.68	1.64
	Slice 2 (Medium)	1829 - 1942	0.62	1.55
	Slice 3 (Newest)	1943 - 2057	0.64	1.59
Claude-4.5	Slice 1 (Oldest)	1609 - 1826	0.47	1.19
	Slice 2 (Medium)	1829 - 1942	0.49	1.26
	Slice 3 (Newest)	1943 - 2057	0.52	1.35

turns. It proves that cascading failures in CodeFlowBench are largely indicative of true capability boundaries rather than trivial syntax errors.

E.4 Sensitivity to Topological Ordering

A potential concern regarding the evaluation stability is that different linearizations of the same non-linear dependency graph could lead to arbitrary fluctuations in the APD and APT metrics.

Experimental Setup To empirically verify the stability of our metrics against graph linearization, we sampled 100 problems exhibiting non-linear dependency structures (i.e., those allowing multiple valid topological sorts). For each problem, we generated four distinct, valid topological linearizations and evaluated the models across these variants.

Results The results, summarized in Table 14, demonstrate that our structural metrics are remarkably robust to task reordering. Statistical analysis reveals that the variance of Average Pass Turn (APT) across topological variants remains exceptionally low (with Standard Deviations ≈ 0.04), while the fluctuation in Average Pass Depth (APD) is even more negligible.

This quantitative stability confirms that both metrics serve as consistent and reliable indicators of model performance. Although different valid linearizations alter the immediate preceding context, they primarily shift the position of parallel subproblems without destabilizing the overall evaluation ranking or obscuring the models' true iterative capabilities.

E.5 Error Case Study

We have defined three typical error types for test in multi-turn pattern, in this part we will introduce several example to illustrate them.We have simplify and arrange model's outputs to make it clear to read. The content of "generated" field is model's output. The content of "harness_result" field is the verification result by running it with test cases."1" denotes accepted, "0" denotes wrong answer and "wrong" denotes running error.

F Impact Statement

This paper introduces CodeFlowBench, a comprehensive benchmark for evaluating code generation models in realistic multi-turn, dependency-driven development scenarios. For research, CodeFlowBench fills a critical gap by providing a standardized suite of tasks that require iterative reasoning, function dependency management, and end-to-end solution assembly. By exposing models' deficiencies in global awareness, instruction consistency, and dependency handling, CodeFlowBench will catalyze the design of new architectures and training paradigms that explicitly model iterative workflows and cross-turn coherence. Its open dataset and evaluation protocol invite the community to develop and compare dependency-aware generation strategies, driving progress toward more robust and developer-friendly code assistants.

In industry, CodeFlowBench offers a practical yardstick for assessing the readiness of AI coding tools in real-world software development, where tasks rarely appear as isolated single-step prompts. Integrating CodeFlowBench into CI/CD pipelines can help organizations detect and remediate weaknesses in model-powered code suggestions before deployment, reducing debugging overhead and technical debt. By highlighting the importance of function reuse, import management, and state consistency across revisions, CodeFlowBench insights can inform best practices for AI-augmented coding workflows, accelerating adoption of reliable co-programming solutions. There are broader societal implications in enabling safer, more maintainable AI-generated code, yet none that we believe warrant special emphasis here.

”problem-id”: ”1799F”,
"subproblems":
[
{"name": "comb",…},
{"name": ”sub",…},
{"name": ”solve_case”,
"statement": "The solve_case function determines the minimum cost of an array after performing the allowed operations. It sorts the array in descending order and iterates over possible numbers of elements to which both operations are applied. For each scenario, it calculates the cost by applying the half and sub operations optimally. The function uses dynamic programming to efficiently compute the minimum cost, ensuring that the constraints on the number of operations are respected.",
"depth": 1,
"dependencies": ["half", "sub"],
"test_code" : [{“input”: “(3, 2, 1, 1, [9, 5, 3])”, “output”: “11” }, …],
"generated" :

```
def solve_case(n: int, b: int, k1: int, k2: int, a: List[int]) -> int:
        # Sort in descending order to apply operations on the largest elements first
        a_sorted = sorted(a, reverse=True)
        min_cost = float('inf')
        # t = number of elements receiving both operations
        max_t = min(k1, k2, n)
        for t in range(max_t + 1):
            rem_k1 = k1 - t
            rem_k2 = k2 - t
            temp = a_sorted.copy()
            # Apply both operations to the first t elements
            for i in range(t):
                temp[i] = sub(half(temp[i]), b)
            # Build list of potential gains for remaining elements
            gains = []
            for i in range(t, n):
                orig = temp[i]
                gain_half = orig - half(orig)
                gain_sub  = orig - sub(orig, b)
                best_gain = max(gain_half, gain_sub)
                gains.append((best_gain, gain_half, gain_sub, i))
            # Prioritize by largest possible reduction
            gains.sort(reverse=True, key=lambda x: x[0])
            used1 = used2 = 0
            for best_gain, g1, g2, idx in gains:
                if used1 < rem_k1 and used2 < rem_k2:
                    if g1 >= g2:
                        temp[idx] = half(temp[idx])
                        used1 += 1
                    else:
                        temp[idx] = sub(temp[idx], b)
                        used2 += 1
                elif used1 < rem_k1:
                    temp[idx] = half(temp[idx])
                    used1 += 1
                elif used2 < rem_k2:
                    temp[idx] = sub(temp[idx], b)
                    used2 += 1
                else:
                    break
            # Compute total cost for this split
            current_cost = sum(temp)
            min_cost = min(min_cost, current_cost)
        return min_cost
```

”harness_result" : [1,1,1,1,1,1,0]}
]

Figure 17: Example of an Incomplete Reasoning (IR) Error by Deepseek-V3.The original problem is 1799F.The algorithm produced by the model fails to find the global optimum because it greedily sorts each element according to the single-step gain—choosing either the halving operation or the subtraction operation in isolation. In reality, achieving the global optimum requires a two-dimensional dynamic-programming solution that considers both operations jointly.

Table 13: Comparison of Absolute Solved Turns (AST) under Standard vs. Non-stop Evaluation

Model	Fail-Stop	Non-Stop	Performance Gain (Δ)			
			Overall	2-Turns	3-Turns	4-Turns
Gemini-3	1.19	1.35	+0.16	+0.08	+0.35	+0.25
GPT-5	0.80	0.87	+0.07	+0.07	+0.10	+0.00
Claude-4.5	0.79	0.87	+0.08	+0.09	+0.22	+0.11

Table 14: Comparison of APD and APT Metrics across Topological Variants

Variant	Gemini-3		GPT-5		Claude-4.5	
	APD	APT	APD	APT	APD	APT
Variant 1	0.86	1.30	0.57	1.11	0.59	1.13
Variant 2	0.86	1.33	0.59	1.22	0.63	1.18
Variant 3	0.87	1.40	0.58	1.17	0.59	1.13
Variant 4	0.84	1.40	0.56	1.10	0.56	1.07
Mean (APT)	1.3575		1.1500		1.1275	
Std. Dev. (APT)	0.0437		0.0461		0.0450	


```
"problem-id": "2005D",
"subproblems":
[
    {"name": " compute_prefix_suffix",
     "statement": "Compute the prefix and suffix GCDs for the given array. The prefix GCD at index i is the GCD of all elements from the start of the array up to index i. Similarly, the suffix GCD at index i is the GCD of all elements from index i to the end of the array. This function is essential for determining the GCDs of subarrays, which is crucial for finding the optimal swap range to maximize the sum of GCDs.",
     "depth": 1,
     "dependencies": [],
     "test_code" : [ "input": "([11, 4, 16, 17, 3, 24, 25, 8], 8)",
            "output": "([0, 11, 1, 1, 1, 1, 1, 1, 1], [0, 1, 1, 1, 1, 1, 1, 1, 8, 0])"}, … ],
     "generated" :
     def compute_prefix_suffix(arr, n):
            prefix_gcd = [0] * (n + 2)
            suffix_gcd = [0] * (n + 2)
            # Compute prefix GCD
            prefix_gcd[0] = 0  # No elements before index 0
            if n >= 1:
                prefix_gcd[1] = arr[0]
                for i in range(2, n + 1):
                    prefix_gcd[i] = gcd(prefix_gcd[i - 1], arr[i - 1])
            # Compute suffix GCD
            suffix_gcd[n + 1] = 0  # No elements after index n
            if n >= 1:
                suffix_gcd[n] = arr[n - 1]
                for i in range(n - 1, 0, -1):
                    suffix_gcd[i] = gcd(suffix_gcd[i + 1], arr[i - 1])
            return (prefix_gcd, suffix_gcd)
    "harness_result" : ["wrong","wrong", "wrong","wrong",…]}
        ]
```


Figure 18: Example 1 of an Insufficient Globalization(IG) error by Deepseek-V3.The original problem is 2005D.The model's generated code omitted the import `math` statement, resulting in an error when calling `gcd`. This issue stems from improper handling of external imports.

```
”problem-id“: ”1734F’,
"subproblems":
[
    {"name": “popcount”,…}
    {"name": “solve”,…}
    {
     "name": ” main”,
    "statement": "Read input, process multiple test cases, and output the Hamming distance for each test case. The function reads the number of test cases and for each test case, it reads the values of `n` and `m`, clears the cache, and computes the Hamming distance using the `solve` function. The results are then printed for each test case.",
     "depth": 0,
     "dependencies": [“solve”],
     "test_code" : [{ "input": "['6\n1 1\n5 10\n34 211\n73 34\n19124639 56348772\n12073412269     96221437021']",
        "output": ["1\n6\n95\n20\n28208137\n48102976088\n”]}, …],
     "generated" :
        import sys
        def main():
            input = sys.stdin.read().split()
            ptr = 0
            T = int(input[ptr])
            ptr += 1
            for _ in range(T):
                n = int(input[ptr])
                m = int(input[ptr + 1])
                ptr += 2
                print(solve(n, m))
    "harness_result" : ["wrong",…]}
        ]
```


Figure 19: Example 2 of an Insufficient Globalization(IG) error by Deepseek-V3.The original problem is 1734F Because the program reads new inputs and performs fresh calculations for each test case, failing to clear the cache beforehand can inadvertently reuse stale results from a previous case. This leads to incorrect output due to unintended interaction between caching and I/O.

”problem-id“: ”1575H’,
"subproblems":
[
{"name": “build_transition”,
“generated”;
```
def build_transition(b):
    m = len(b)
    prefix = compute_prefix(b)
    transition = [{} for _ in range(m)]
    match_counts = [0] * m
    for state in range(m):
        for c in ['0', '1']:
            if state < m and c == b[state]:
                next_state = state + 1
            else:
                next_state = 0
            if state > 0:
                j = prefix[state - 1]
                while j > 0 and c != b[j]:
                    j = prefix[j - 1]
                if c == b[j]:
                    next_state = j + 1
                else:
                    next_state = 0
            transition[state][c] = next_state
        # The match count for the final state is 1
        if state == m - 1:
            match_counts[state] = 1
    return transition, match_counts
```
…}
{
"name": ” min_changes”,
"statement": "Compute the minimum number of changes required in the string `a` to have exactly `k` occurrences of the string `b` as a substring. This function uses dynamic programming with three states: position in `a`, position in `b`, and the number of matches left. The transition table built by `build_transition` is used to efficiently compute the state transitions.",
"depth": 1,
"dependencies": [“build_transition”],
"test_code" : ["input": "('100101011', '101', 0)","output": "1"}, …],
"generated" :
```
def min_changes(a, b, k):
    n = len(a)
    m = len(b)
    if m == 0 or m > n:
        return 0 if k == 0 else -1  # edge cases
    transition, match_counts = build_transition(b)
    max_matches = n - m + 1
    # Initialize DP table: dp[state][matches] = min changes
    INF = float('inf')
    dp_prev = [[INF] * (max_matches + 2) for _ in range(m + 2)]
    dp_prev[0][0] = 0  # initial state: 0 matches, state 0, 0 changes
    for i in range(n):
        dp_next = [[INF] * (max_matches + 2) for _ in range(m + 2)]
        current_char = a[i]
        for state in range(m + 1):
            for matches in range(max_matches + 1):
                if dp_prev[state][matches] == INF:
                    continue
                # Try both possible characters ('0' and '1')
                for c in ('0', '1'):
                    new_state = transition[state][c] if state < m else 0
                    new_matches = matches
                    if new_state == m:
                        new_matches += 1
                        if new_matches > max_matches:
                            continue  # cannot exceed max_matches
                    cost = dp_prev[state][matches] + (0 if c == current_char else 1)
                    if cost < dp_next[new_state][new_matches]:
                        dp_next[new_state][new_matches] = cost
        dp_prev = dp_next
    # After processing all characters, find the minimal cost for exactly k matches
    min_cost = INF
    for state in range(m + 1):
        if dp_prev[state][k] < min_cost:
            min_cost = dp_prev[state][k]
    return min_cost if min_cost != INF else -1
```
“harness_result” : [1,0,0,0,1,1,1]}
]

Figure 20: Example of an Instruction Misinterpretation(IM) error by Deepseek-V3.The original problem is 1575H Although the DP routine correctly unpacks both transition and match_counts from build_transition(b), it merely treats match_counts as an ordinary list. In reality, match_counts carries two essential pieces of semantic information.(1)Match indicator.It denotes a complete match of b.(2)Backtrack hint. It’s a hint in combination with the prefix function, indicating how far the automaton should jump back after a match to continue detecting overlapping occurrences. This error is caused by models’ misinterpretation to the dependency relationship between tool function and top-level function