DC-Coder: A Dual-Consensus Multi-Agent Framework for Complex Code Generation

Anonymous ACL submission

Abstract

While Large Language Models have advanced code generation, they struggle with complex tasks due to over-reliance on inaccurate self-generated test cases. To address this, we propose DC-Coder, a novel dual-consensus multi-agent framework. At the planning level, the framework synthesizes a global consensus plan by resolving conflicts among multiple parallel candidate strategies, effectively eliminating reasoning blind spots. At the execution level, it constructs a function dependency tree as a structural constraint and leverages testing agents for generalized input generation. By identifying behavioral similarities across candidate codes, execution consensus is formed that ensures both structural integrity and functional correctness. Extensive experiments on five competition-level benchmarks demonstrate that DC-Coder consistently outperforms strong baselines like MapCoder, establishing a new paradigm for automated code generation.

1 Introduction

In recent years, Large Language Models have achieved transformative breakthroughs in the field of automated code generation (Li et al., 2022). Although these models excel at synthesizing simple code snippets, they continue to face substantial challenges when confronted with competition level programming tasks that demand complex reasoning, multi-step logical constraints, and strict alignment with human intent (Austin et al., 2021; Chowdhery et al., 2023; Nijkamp et al., 2022). Traditional generative strategies are mainly based on single path reasoning paradigms (Chen et al., 2021; Wei et al., 2022). This approach often exhibits limitations. Any minor deviation from the initial logical deduction steps can trigger error propagation, which can lead to functional failure of the synthesized programs (Ridnik et al., 2024; Parvez et al., 2021).

To overcome the limitations of single agent zero shot reasoning (Shum et al., 2023; Zhang et al., 2022), the research community has actively explored multi-agent collaborative frameworks and self reflection mechanisms based on execution feedback (Shinn et al., 2023; Chen et al., 2022). In addition, divide and conquer strategies have been introduced to decompose intricate programming tasks into manageable sub functions (Jiang et al., 2024). However, these existing methodologies still have room for further optimization in practical applications.

At the planning layer, most multi-agent systems depend on a singular generative heuristic trajectory (Islam et al., 2024). They lack a systematic mechanism to parallelize multi view strategy generation and reconcile logical conflicts. Relying on a singular perspective frequently leads to blind spots in reasoning, making it difficult to capture overlooked edge cases in highly complex algorithms (Yasunaga et al., 2023).

At the execution layer, current modular generation typically follows a bottom up assembly paradigm. Although recent work has effectively utilized execution based verification to filter errors in sub functions (Chen et al., 2024), this fine grained decomposition frequently fragments the global context. Synthesizing complex code in isolated segments leads to interface contract misalignments and structural hallucinations when integrated into a unified program scope. Furthermore, requiring models to generate reliable unit tests for intermediate auxiliary functions without explicit expected output frequently triggers test hallucinations, incurring notable computational overhead (Huang et al., 2023; Zhou et al., 2023; Pacheco et al., 2007). Such overhead significantly restricts the practical deployment of autonomous coding systems in resource constrained environments.

To address these challenges, we propose a multi-agent code generation framework, abbrevi-

ated DC-Coder. Inspired by standard human software development workflows, DC-Coder coordinates four specialized agents, namely the Planner, the Architect, the Coder, and the Tester. This collaboration achieves high quality code synthesis through a dual consensus mechanism. To address the fragility of the planning layer, we introduce a Planning Consensus mechanism in which the Planner generates multiple candidate plans across diverse dimensions and synthesizes a global consensus plan through conflict resolution between plans, effectively neutralizing individual reasoning biases (Wen et al., 2026). To resolve context fragmentation at the execution layer, we introduce a holistic generation paradigm guided by a structural skeleton. The Architect first generates a dependency tree consisting only of function signatures and declarative definitions based on the consensus plan. Subsequently, the Coder performs holistic code generation in a single pass under the strong structural constraints of the dependency tree. This architecture inherits the robustness of functional consensus while avoiding interface mismatches. Finally, the system executes majority voting based on behavioral consistency in an isolated sandbox using global test input to identify the functionally correct final program. Figure 1 shows an overview of our approach.

We conducted thorough experiments across five mainstream programming synthesis benchmarks. These include basic programming datasets such as HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021), as well as challenging competition level problem solving benchmarks such as APPS (Hendrycks et al., 2021), xCodeEval (Khan et al., 2024), and CodeContests (Li et al., 2022). By leveraging fully open source foundational models and explicitly enabling their native reasoning paths, we evaluate the framework under compute matched conditions. Our approach consistently improves problem solving capabilities, achieving highly competitive performances and outperforming strong baselines such as MapCoder (Islam et al., 2024). Moreover, our method delivers stable performance across various problem difficulties. We also provide detailed ablation studies and qualitative analyzes to further illustrate the effectiveness of the dual consensus mechanism and the holistic generation paradigm. These empirical findings suggest that integrating strategic planning with micro structural execution fundamentally enhances the reasoning capacity of language models.

2 Related Work

2.1 Large Language Models for Code Generation

Language models demonstrate impressive capabilities in automating code generation (Guo et al., 2024; Hui et al., 2024; Yang et al., 2025; Touvron et al., 2023; Roziere et al., 2023; Jiang et al., 2023; Li et al., 2023). Although proficient in simple snippets (Fried et al., 2022; Chen et al., 2021; Austin et al., 2021; Nijkamp et al., 2022; Allal et al., 2023), they struggle with competition level programming that requires complex reasoning (Li et al., 2022). Prompt strategies such as Chain of Thought (Wei et al., 2022) and self-planning (Jiang et al., 2024) mitigate this by decomposing problems into intermediate steps. However, these single path methods remain highly susceptible to cascading errors (Yao et al., 2023).

2.2 Multi-agent Systems and Task Planning

To address single agent limitations, research explores multi-agent collaborative frameworks that assign specialized roles (Islam et al., 2024; Dong et al., 2024; Islam et al., 2025; Pan et al., 2025). For strategic planning, advanced approaches recover past experiences to generate multiple candidate plans (Wen et al., 2026; Yasunaga et al., 2023; Parvez et al., 2021). Summarizing discrepancies among these candidates synthesizes a consensus plan that effectively eliminates logical conflicts. While macro-level consensus improves planning, translating high level plans directly into monolithic programs often exceeds model reasoning capacities.

2.3 Code Decomposition and Execution Based Verification

To manage complex requirements, divide and conquer strategies branch complex tasks into smaller sub functions (Chen et al., 2024; Zhang et al., 2023). While reducing localized complexity, this introduces the risk of error propagation. Traditional methods correct errors using feedback from model generated unit tests (Ridnik et al., 2024; Chen et al., 2023, 2022; Le et al., 2022), but these assertions often remain unreliable. To bypass this dependency, functional consensus mechanisms execute multiple sampled implementations against diverse inputs and evaluate behavioral consistency to design optimal functions (Chen et al., 2024).

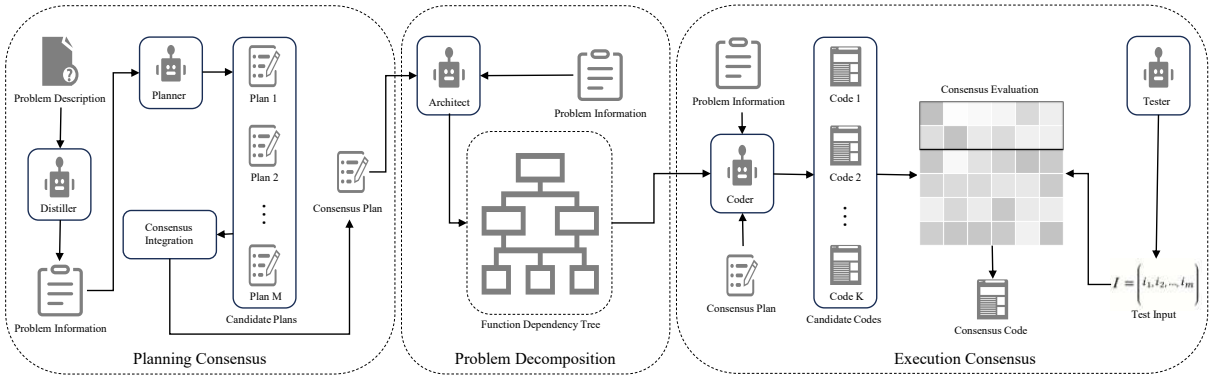


Figure 1: Overview of the DC-Coder framework. The pipeline consists of three core phases: Planning Consensus, Problem Decomposition, and Execution Consensus.

3 Methods

3.1 Problem Formulation

Following the standard paradigm of execution-based program synthesis (Chen et al., 2021; Hendrycks et al., 2021), a code generation task is formally defined as a tuple $T = (X, E_{vis}, E_{hid})$. Here, X denotes the description of natural language problems, $E_{vis} = \{(i_j, o_j)\}_{j=1}^{N_v}$ represents a set of visible test cases (input-output pairs) provided in the prompt, and E_{hid} denotes the hidden test cases reserved strictly for the final evaluation. Our primary objective is to generate an executable target program Y that satisfies the functional correctness mapping:

$$\forall (i, o) \in E_{vis} \cup E_{hid}, \quad \text{Exec}(Y, i) = o \qquad (1)$$

where $\text{Exec}(\cdot)$ is a deterministic execution function running in an isolated sandbox environment.

3.2 Framework Overview

In order to overcome the cascading illusion caused by the vulnerability of single path inference in large language models when dealing with complex programming tasks, as well as the problem of inaccurate self generated test cases, we propose a multi-agent code generation framework. This framework is driven by a highly collaborative professional multi-agent system, which encompasses four core roles: planner, architect, encoder, and tester. Its workflow is systematically divided into three key stages: Planning Consensus, Plan-Driven Decomposition, and Execution Consensus.

3.3 Planning Consensus

The reasoning process begins with the Planner Agent to establish a strategic direction to solve the complex problem X. To acquire diverse perspectives on problem-solving without introducing the computational overhead of external retrieval, the Planner Agent leverages directly its internal knowledge to generate a set of M diverse candidate plans $\mathcal{P} = \{p_1, p_2, ..., p_M\}$ in parallel (Wen et al., 2026).

To formulate a reliable and comprehensive strategy, the Planner Agent performs cross-plan conflict resolution. Identify overlapping logical steps and rectify deviations between candidate plans, synthesizing them into a unified consensus plan p_{con}. This process is formally defined as:

$$p_{con} = \text{LLM}_{planner_consensus}(X, \mathcal{P}) \qquad (2)$$

The synthesized p_{con} captures the essential algorithmic logic and critical edge cases, providing global guidance for the subsequent stages while effectively bypassing the fragility of single-path reasoning.

3.4 Plan-Driven Decomposition

Translating a high-level plan directly into a monolithic program often leads to structural errors. Therefore, guided by X and p_{con}, the Architect Agent performs structural decomposition to construct a functional dependency tree $G = (V, E)$ (Chen et al., 2024).

Each node $v \in V$ represents a sub-module within the target program, mathematically defined as:

$$v = \langle s_v, d_v, D_v \rangle \qquad (3)$$

where s_v denotes the function signature, d_v represents the docstring detailing the module's expected behavior, and D_v specifies its set of child dependencies. At this stage, all function nodes

contain only stubs (empty definitions) without concrete implementations. The dependency tree G serves as a strictly defined structural skeleton, effectively constraining the LLM search space through clear modular boundaries.

3.5 Execution Consensus

To overcome the severe context fragmentation caused by the fine-grained bottom-up assembly, the Coder Agent performs skeleton-guided holistic generation. Constrained simultaneously by the problem description X, the consensus plan p_{con}, and the structural skeleton G, the Coder Agent synthesize k complete candidate programs $\mathcal{C} = \{c^1, c^2, ..., c^k\}$ in parallel:

$$\mathcal{C} = \text{LLM}_{coder}(X, p_{con}, G, k) \tag{4}$$

This holistic generation paradigm ensures that the interface contracts between parent and child functions are perfectly aligned within a unified context window.

Finally, we introduce an execution-based functional consensus mechanism to determine the optimal solution. To mitigate the risk of *systematic false positive consensus*where multiple flawed candidates share identical logical blind spotsthe Tester Agent employs targeted test generation strategies. Rather than relying solely on surface-level problem descriptions, it incorporates boundary value analysis and stress testing to synthesize a robust set of global test inputs I_{test}.

All candidate programs are executed against I_{test} in an isolated sandbox. We determine the optimal target program Y by majority voting by computing the behavioral consistency (i.e., output equivalence) among the candidates:

$$Y = \arg\max_{c \in \mathcal{C}} \sum_{i \in I_{test}} \sum_{\substack{c' \in \mathcal{C} \\ c' \neq c}} \mathbb{I}\left(\text{Exec}(c, i) = \text{Exec}(c', i)\right) \tag{5}$$

where $\mathbb{I}(\cdot)$ is the indicator function. Furthermore, to prevent the consensus mechanism from favoring computationally inefficient brute-force solutions, any candidate that triggers a Time-Limit Exceeded (TLE) or runtime error during sandbox execution is heavily penalized by assigning its validation score to $-\infty$. Through this rigorous global execution consensus, we select the functionally most robust implementation, effectively filtering out candidates afflicted by hallucinations, logical flaws, or algorithmic inefficiencies.

4 Experiments

4.1 Experimental Setup

Models. In contrast to previous studies relying heavily on closed source models, we deployed two of the most commonly used open source large language models, DeepSeek-R1-Distill-Qwen-7B (Guo et al., 2025) and Qwen3-8B (Yang et al., 2025), as our foundational backbone networks.

Datasets. To comprehensively evaluate the code generation capability of our proposed framework, we conduct experiments on five widely adopted benchmark datasets spanning both basic and competitive programming domains. The two basic programming datasets are HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021). The problem set sizes for HumanEval and MBPP are 164 and 397, respectively. To assess the models in complex problem solving tasks, we utilize three competition level programming datasets, namely the Automated Programming Progress Standard(APPS) (Hendrycks et al., 2021), xCodeEval (Khan et al., 2024), and CodeContests (Li et al., 2022). For our evaluation, we used 150, 106 and 165 problems from these datasets, respectively.

Evaluation Metrics. We adopt the Pass@1 evaluation metric, which the model is considered successful only if the single generated solution is functionally correct and passes all hidden test cases (Chen et al., 2021). All experiments are conducted with a single run, and the reported results are based on these individual executions.

4.2 Baselines

We compare our proposed method against four strong baselines and state of the art multi-agent approaches. Direct Prompting relies purely on the inherent zero shot capabilities of the language model by instructing it to generate code without any explicit guidance (Kojima et al., 2022). Chain of Thought generates a thought path by decomposing problems into step by step solutions, improving the accuracy and efficiency of solving complex tasks (Wei et al., 2022). Reflexion introduces a self reflection based strategy that gathers feedback to refine the models, allowing them to self correct previous imperfect responses (Shinn et al., 2023). MapCoder represents a state of the art framework in which different agents are assigned roles as retriever, planner, coder, and debugger to cooperatively generate code (Islam et al., 2024).

LLM	Approach	Simple Problems		Contest-Level Problems		
		HumanEval	MBPP	APPS	xCodeEval	CodeContests
Qwen3-8B	Direct	36.59	20.65	3.33	18.87	1.82
	CoT	81.10	70.28	2.67	15.09	3.64
	Reflexion	87.80	78.59	-	-	-
	MapCoder	87.20	83.63	10.67	33.02	17.58
	Ours	**92.07**	**87.66**	**12.67**	**48.11**	**20.00**
DeepSeek-R1-Distill-Qwen-7B	Direct	70.12	60.20	2.67	19.81	2.42
	CoT	75.00	67.51	2.00	0.94	2.42
	Reflexion	82.93	64.99	-	-	-
	MapCoder	82.93	71.54	6.00	42.45	12.73
	Ours	**83.54**	**75.06**	**12.67**	**50.00**	**14.55**

Table 1: Pass@1 performance of different LLMs and approaches on basic and competitive code generation benchmarks. The best results for each LLM are highlighted in bold.

CodeContest (Pass@5)		
Approach	**Qwen3-8B**	**DeepSeek-R1-Distill-Qwen-7B**
Direct	6.06%	3.64%
CoT	3.03%	5.45%
Ours	**20.61%**	**16.36%**

Table 2: Pass@5 results on the CodeContest dataset.

4.3 Performances on basic and competitive code generation

Based on the comprehensive evaluation across five main benchmark datasets presented in Table 1, our method achieves state-of-the-art Pass@1 performance on all datasets, significantly outperforming strong baselines including Direct Prompting, Chain-of-Thought (CoT), Reflexion, and the multi-agent framework MapCoder.

In basic programming tasks, when using Qwen3-8B as the base model, our method achieves accuracies of 92.07% and 87.66% on the HumanEval and MBPP datasets, respectively. In comparison, the advanced multi-agent collaborative method MapCoder yields 87.20% and 83.63% on these benchmarks. These results indicate that our method maintains a consistent performance advantage in handling fundamental code synthesis. Similar improvements are observed when using the DeepSeek-R1-Distill-Qwen-7B model, reaching 83.54% on HumanEval and 75.06% on MBPP.

Our method demonstrates an overwhelming advantage on competition-level complex tasks such as APPS, xCodeEval, and CodeContests, which require intricate multi-step reasoning. For example, on the notoriously difficult APPS dataset, the Pass@1 accuracy of Direct Prompting with Qwen3-8B is merely 3.33%, whereas our method substantially elevates it to 12.67%. Furthermore, our framework reaches 48.11% on xCodeEval and 20.00% on CodeContests with Qwen3-8B, far exceeding traditional baseline methods. When equipped with DeepSeek-R1-Distill-Qwen-7B, our method achieves 12.67%, 50.00%, and 14.55% on APPS, xCodeEval, and CodeContests, respectively. This strongly proves that, when confronted with complex logical constraints, our dual-consensus mechanism effectively overcomes the fragility of single-path reasoning to generate functionally correct and robust programs.

To further evaluate the stability of code generation, we report the Pass@5 performance on the challenging CodeContest dataset. As shown in Table 2, our method achieves 20.61% accuracy when utilizing Qwen3-8B, significantly outperforming Direct Prompting and Chain-of-Thought (CoT), which obtain 6.06% and 3.03% respectively. With the DeepSeek-R1-Distill-Qwen-7B backbone, our method reaches 16.36% accuracy, similarly demonstrating a substantial improvement over the baseline methods (3.64% for Direct and 5.45% for CoT). These results confirm that our method not only improves the success rate of single-path generation but also exhibits excellent robustness in multi-sampling scenarios.

4.4 Performance with Varying Difficulty Levels

To investigate the robustness of the model across difficulty gradients, Figure 2 and Figure 3 detail the performance distributions on the APPS and xCodeEval datasets, respectively. As task diffi-

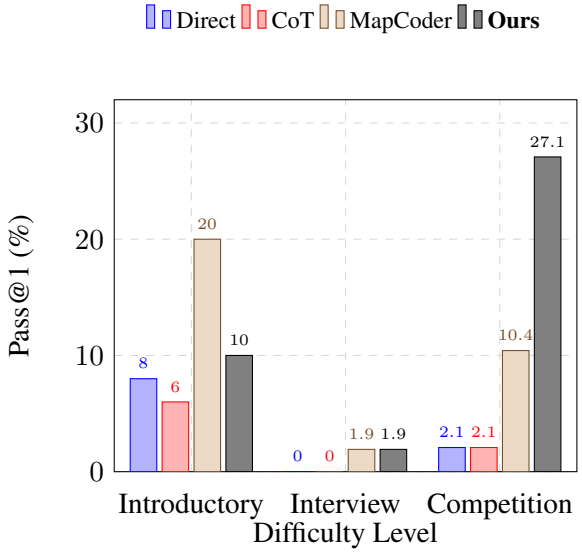


Figure 2: Comparison of Pass@1 accuracy on the APPS dataset across three difficulty levels.

LLM	Approach	HumanEval	CodeContest
Qwen3-4B	Direct	90.85%	18.18%
	CoT	86.59%	15.15%
	Ours	**90.85%**	**29.70%**

Table 3: Pass@1 results using Qwen3-4B.

culty escalates, traditional baselines suffer a cliff-like performance drop. Specifically, in APPS Interview-level tasks, Direct Prompting and Chain-of-Thought accuracies plummet to zero, whereas our method and MapCoder maintain 1.9%. In the highly challenging Competition-level tasks, our method achieves 27.1% accuracy, significantly surpassing MapCoder's 10.4% and demonstrating exceptional resistance to performance degradation. Similarly, xCodeEval results reveal that our method substantially outperforms Direct Prompting across core algorithm types, notably Implementation, Math, and Brute Force. Furthermore, while the efficacy of Direct Prompting sharply declines for problems rated above 900, our framework consistently resolves highly difficult tasks within the 1000 to 1500 rating interval, further corroborating its robust generalization capability in complex, long-tail scenarios.

4.5 Performance Across Different LLMs

To evaluate robustness across parameter scales, Table 3 and Table 4 detail the results for Qwen3-4B and Qwen3-0.6B. With Qwen3-4B, our method matches the best baseline on HumanEval and significantly leads Direct Prompting and Chain-of-Thought on the challenging CodeContest with a precision of 29.70%. Crucially, on the extremely small Qwen3-0.6B, our method achieves substantial improvements on both HumanEval and the complex APPS dataset, nearly tripling the baseline performance on the latter. These findings demonstrate that our architecture effectively stimulates the reasoning potential of base models across varying scales, yielding consistent gains in both fundamental and complex programming tasks.

LLM	Approach	HumanEval	APPS
Qwen3-0.6B	Direct	42.07%	0.67%
	CoT	44.51%	0.67%
	Ours	**54.27%**	**2.00%**

Table 4: Pass@1 results using Qwen3-0.6B.

Planning Consensus	Modular Decomp.	Execution Consensus	Pass@1	Performance Drop
×	×	×	87.80%	4.27%
×	✓	✓	88.41%	3.66%
✓	✓	×	87.20%	4.87%
✓	×	✓	87.20%	4.87%
✓	✓	✓	92.07%	-

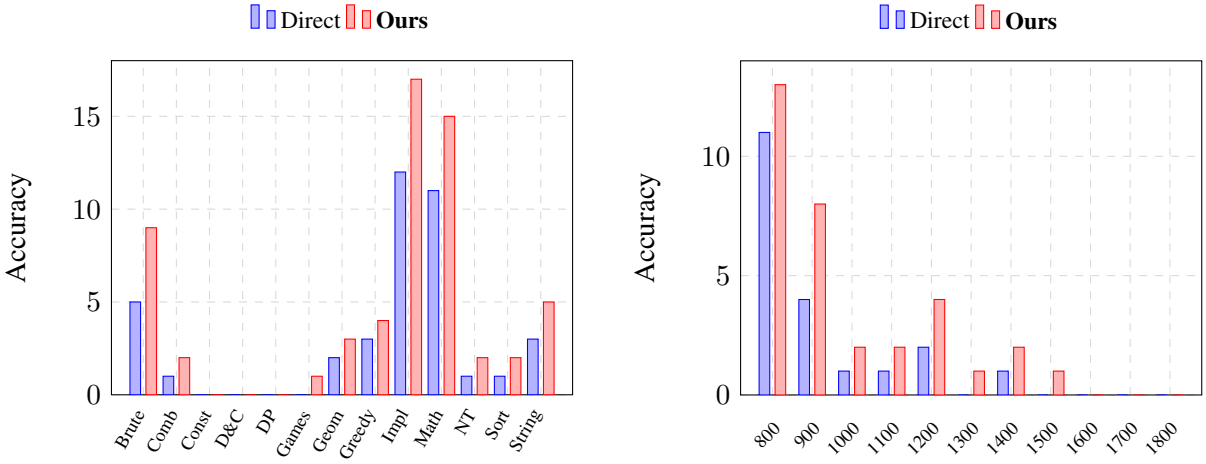
Table 5: Ablation study of DC-Coder on the HumanEval dataset using Qwen3-8B.

4.6 Ablation Study

To verify the effectiveness of the component, Table 5 details the ablation results on the HumanEval dataset. Removing all components drops the accuracy to 87.80% with a 4.27% performance decrease. Omitting only the planning consensus reduces accuracy to 88.41%, while lacking either execution consensus or modular decomposition lowers it to 87.20%. The complete architecture integrating all three core modules achieves the optimal 92.07% accuracy. These findings confirm that planning consensus, modular decomposition, and execution consensus are strictly indispensable. Their deep synergy constitutes the solid foundation for the superior code generation performance of our framework.

4.7 Impact of k and M

To investigate the impact of the number of consensus plans and code candidates generated on the final performance, we performed a hyperparameter sensitivity analysis. As shown in Table 6, when the number of consensus plans is fixed at three, increasing the candidate count from one to three



(a) Algorithm Types (b) Difficulty Levels

Figure 3: Detailed performance analysis on the xCodeEval dataset. (a) Number of correctly solved problems categorized by algorithm types (tags). (b) Distribution of correct answers across different difficulty levels (800–1800).

Plan Count (M)	**Candidate Count** (k)		
	1	**3**	**5**
3	87.20%	92.07%	87.80%
5	90.24%	86.59%	85.98%

Table 6: Pass@1 results on HumanEval by varying the number of consensus plans (M) and generated code candidates (k).

Dataset	**Method**		
	Direct	**CoT**	**Ours**
LiveCodeBench	13.00%	12.00%	**16.00%**
BigCodeBench	10.89%	10.89%	**15.84%**

Table 7: Pass@1 results on LiveCodeBench and BigCodeBench.

significantly improves the precision of our method on the HumanEval dataset from 87.20% to the optimal 92.07%. However, further increasing the candidate count to five leads to a performance decline to 87.80%. Furthermore, when the number of consensus plans increases to five, the overall performance is generally inferior to the configuration with three consensus plans, peaking at only 90.24%. This indicates that a moderate number of plans effectively synthesizes multi-perspective logic and filters out noise, whereas an excessively large search space may introduce redundancy and conflicts, disrupting the execution-level consensus decision. Consequently, setting both the consensus plan count and the candidate code count to three achieves the optimal balance between generation diversity and global structural consistency.

4.8 Evaluation on other benchmarks

To further evaluate the applicability of the framework in broader and more challenging practical programming scenarios, we conducted extended tests on the LiveCodeBench(Jain et al., 2024) and BigCodeBench(Zhuo et al., 2024) datasets. As shown in Table 7, our method achieves the optimal Pass@1 accuracy on both benchmarks. On the LiveCodeBench dataset, our method reaches an accuracy of 16.00%, which is significantly higher than the 13.00% of the Direct Prompting method and the 12.00% of the Chain-of-Thought method. On the BigCodeBench dataset, our method achieves an accuracy of 16.83%, likewise substantially outperforming the 10.89% of Direct Prompting and the 10.89% of Chain-of-Thought. These experimental results fully indicate that our method not only excels in standard algorithmic tasks but also maintains robust generation capabilities and high adaptability when faced with diverse and complex programming instructions that closely resemble real-world development environments.

4.9 Efficiency analysis

To comprehensively evaluate the trade-off between computational cost and performance improvement, we compare the differences in the average number of API calls and token consump-

LLM	Dataset	Average for Ours		Average for Direct Prompting		Accuracy Gain
		API Calls	Tokens (k)	API Calls	Tokens (k)	
Qwen3-8B	HumanEval	9.42	38.21	1	3.27	+55.48%
	MBPP	9.57	35.02	1	2.81	+67.01%
	APPS	12.41	85.68	1	4.73	+9.34%
	xCodeEval	11.69	83.94	1	4.84	+29.24%
	CodeContests	12.59	104.09	1	5.12	+18.18%
	Average	**11.14**	**69.39**	**1**	**4.15**	**+35.85%**
DeepSeek-R1-Distill-Qwen-7B	HumanEval	9.66	35.45	1	2.85	+9.15%
	MBPP	9.78	29.28	1	2.23	+21.66%
	APPS	12.56	88.05	1	4.37	+1.33%
	xCodeEval	11.94	78.35	1	4.35	+16.98%
	CodeContests	12.73	103.33	1	4.74	+6.06%
	Average	**11.33**	**66.89**	**1**	**3.71**	**+11.04%**

Table 8: Average number of LLM calls and thousands of tokens used per problem. The table illustrates the efficiency and performance trade-off between Direct Prompting and our DC-Coder framework.

tion between Direct Prompting and our method. As shown in Table 8, multi-agent collaboration inevitably introduces additional reasoning and interaction overhead. When using the Qwen3-8B model, our method consumes an average of 69.39 thousand tokens and makes 11.14 API calls per problem, representing an increase in system overhead compared to the 4.15 thousand tokens and 1 API call consumed by Direct Prompting. This computational investment yields an average accuracy gain of 35.85%. In particular, on basic benchmarks such as MBPP, the accuracy improvement reaches 67.01%. A similar trend is observed when using the DeepSeek-R1-Distill-Qwen-7B model, where our method achieves an average performance gain of 11.04% at an average cost of 66.89 thousand tokens and 11.33 API calls. The experimental data demonstrate that while our method improves code generation accuracy, it requires significantly higher computational resources. The reliance on multi-agent collaboration leads to a notable increase in both API calls and token consumption compared to Direct Prompting. Consequently, future work needs to address the high computational cost of our approach to make it more efficient for practical deployment.

5 Conclusion

In this paper, we introduced DC-Coder, a multi-agent framework that tackles complex programming problems. By integrating macro-level planning consensus with micro-level execution consensus, and utilizing a holistic generation mechanism constrained by a functional dependency tree, our approach overcomes the fragility of single-path reasoning and ensures global structural integrity. Extensive experiments on five benchmarks demonstrate that DC-Coder significantly outperforms state-of-the-art baselines, especially in competition-level tasks. Notably, it achieves these substantial gains, establishing a robust and competitive new paradigm for automated code generation using open-source large language models.

Limitations

Despite its remarkable performance, DC-Coder has a few limitations, the most prominent being its significantly high computational cost. Achieving consensus requires intensive multi-agent interactions, generating multiple candidate plans, and sampling k complete programs in parallel. As demonstrated in our efficiency analysis, this multi-agent collaborative process inherently consumes substantially more inference resources, leading to a drastic increase in both API calls and token consumption compared to single-path zero-shot generation. This computational overhead may limit the framework's real-time applicability and its deployment in resource-constrained environments. Second, the reliability of the execution consensus is heavily based on the test inputs generated by the Tester Agent. Vague problem descriptions may result in inadequate test cases that miss critical edge cases, potentially compromising consensus accuracy.

References

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, and 1 others. 2023. Santacoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988*.

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*.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*.

Jingchang Chen, Hongxuan Tang, Zheng Chu, Qianglong Chen, Zekun Wang, Ming Liu, and Bing Qin. 2024. Divide-and-conquer meets consensus: Unleashing the power of functions in code generation. *Advances in Neural Information Processing Systems*, 37:67061–67105.

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*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, and 1 others. 2023. Palm: Scaling language modeling with pathways. *Journal of machine learning research*, 24(240):1–113.

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2024. Self-collaboration code generation via chatgpt. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–38.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, 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, Yifan 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*.

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, 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*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. Mapcoder: Multi-agent code generation for competitive problem solving. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4912–4944.

Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2025. Codesim: Multi-agent code generation and problem solving through simulation-driven planning and debugging. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 5113–5139.

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*.

Albert Qiaochu Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de Las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7b. *ArXiv*, abs/2310.06825.

Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2024. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–30.

Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. 2024. Xcodeeval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6766–6805.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328.

Jingyao Li, Pengguang Chen, Bin Xia, Hong Xu, and Jiaya Jia. 2023. Motcoder: Elevating large language models with modular of thought for challenging programming tasks. *arXiv preprint arXiv:2312.15960*.

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.

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*.

Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE'07)*, pages 75–84. IEEE.

Ruwei Pan, Hongyu Zhang, and Chao Liu. 2025. Codecor: An llm-based self-reflective multi-agent framework for code generation. *arXiv preprint arXiv:2501.07811*.

Md Rizwan Parvez, Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval augmented code generation and summarization. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 2719–2734.

Tal Ridnik, Dedy Kredo, and Itamar Friedman. 2024. Code generation with alphacodium: From prompt engineering to flow engineering. *arXiv preprint arXiv:2401.08500*.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652.

KaShun Shum, Shizhe Diao, and Tong Zhang. 2023. Automatic prompt augmentation and selection with chain-of-thought from labeled data. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 12113–12139.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, and 1 others. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Chao Wen, Jie Liu, and Liang Du. 2026. Consensus planning boosts llm code generation. *Information and Software Technology*, page 108030.

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*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822.

Michihiro Yasunaga, Xinyun Chen, Yujia Li, Panupong Pasupat, Jure Leskovec, Percy Liang, Ed H Chi, and Denny Zhou. 2023. Large language models as analogical reasoners. *arXiv preprint arXiv:2310.01714*.

Yifan Zhang, Jingqin Yang, Yang Yuan, and Andrew Chi-Chih Yao. 2023. Cumulative reasoning with large language models. *arXiv preprint arXiv:2308.04371*.

Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. 2022. Automatic chain of thought prompting in large language models. *arXiv preprint arXiv:2210.03493*.

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

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*.

A Appendix

A.1 Algorithm

Algorithm 1 shows the pseudo-code of our prompting technique.

A.2 Details Promptings

The detailed prompting of the Distiller Agent, Planner Agent, Architect Agent, Coder Agent, and Tester Agent are shown in Figure 4, 5, 6, 7, 8 and 9 respectively.

A.3 Examples

Figure 10 shows a comparative example of DC-Coder compared to MapCoder in solving problems.

Algorithm 1 DC-Coder

```
M ← number of candidate plans
k ← number of generated code candidates

# Start of Information Extraction
problem_info ← DistillerAgent(Problem)

# Start of Planning Consensus
𝒫 ← empty array of size M
for i ← 1 to M do
    𝒫[i] ← PlannerAgent(problem_info)
end for
p_con ← ConsensusIntegration(problem_info, 𝒫)

# Start of Problem Decomposition
G ← ArchitectAgent(problem_info, p_con)

# Start of Execution Consensus
𝒞 ← empty array of size k
for j ← 1 to k do
    𝒞[j] ← CoderAgent(problem_info, p_con, G)
end for

I_test ← TesterAgent(problem_info)
Y ← null
max_score ← −1

for c in 𝒞 do
    passed, outputs ← Execute(c, I_test)
    if passed then
        score ← 0
        for c′ in 𝒞 do
            _, outputs′ ← Execute(c′, I_test)
            if c ≠ c′ and outputs == outputs′ then
                score ← score + 1
            end if
        end for

        if score > max_score then
            max_score ← score
            Y ← c
        end if
    end if
end for

return Y
```

Distiller Agent

Given a natural language description of a programming problem, perform a rigorous semantic extraction to distill the essential problem information.

Problem Description:
{problem_prompt}

Required Extraction Dimensions:
You must strictly output the distilled information according to the following analytical taxonomy:

1. Objective Formulation
Define the primary computational or optimization objective concisely. Formally describe the exact input/output schema and critical functional requirements.
2. Formal Constraints & Invariants
Extract all explicit operational rules, variable interdependencies, and implicit logical constraints.
3. Boundary Conditions & Degenerate Cases
Identify potential algorithmic vulnerabilities, including extreme edge cases, empty/null states, and asymptotic boundary values that require specialized handling.

Important: You must output exclusively the structured distillation defined above. The generation of any executable code, pseudo-code, or implementation details is strictly prohibited. Do not add conversational filler or extra explanations.

Figure 4: Prompt for Distiller Agent.

Planner Agent

Planning Generation Prompt:
Given a complex competitive programming problem, synthesize a concrete and robust algorithmic plan to solve the problem in {Language}.

Problem to be solved:
{Problem_Info}

Required Planning Dimensions:
You must structure your planning according to the following analytical dimensions:
1. Core Algorithm Identification
Identify the primary algorithmic paradigm (e.g., Dynamic Programming, Greedy Strategy, Graph Traversal, Binary Search).
2. Data Structure Specification
Explicitly define the required data structures to support the algorithmic logic.
3. Step-by-Step Execution Logic
Outline a granular, sequential process to deduce the target output from the given inputs.
4. Edge Cases & Boundary Conditions
Identify potential vulnerabilities, boundary values, and specify the handling mechanisms.

Important: You should synthesize only the strategic planning to solve the problem based on the exact dimensions provided above. You are strictly prohibited from generating any concrete code implementations. Do not add conversational filler, extra explanations, or polite introductory words.

Figure 5: Prompt for Planner Agent.

Planning Consensus

Consensus Integration Prompt:
Given a structured problem specification and a set of {len(plans)} parallel candidate plans derived from diverse heuristic trajectories, your objective is to synthesize a globally optimal consensus plan. You must systematically resolve logical conflicts across the candidate strategies to effectively mitigate individual reasoning biases and eliminate reasoning blind spots.

Problem Specification:
{problem_info}

Candidate Plans:
{plans_text}

Internal Reasoning Directives:
You must implicitly execute the following analytical pipeline prior to formulating your final response:
1. Cross-Plan Conflict Detection
Rigorously contrast the candidate plans to identify disparities in core algorithmic paradigms, data structure selections, and boundary condition handling mechanisms.
2. Resolution Heuristics
Formulate internal directives to reconcile identified contradictions, isolating and retaining the most robust, logically sound, and computationally efficient components from the candidates.
3. Global Consensus Synthesis
Amalgamate the validated components into a singular, cohesive strategic plan that encompasses the exact analytical dimensions required for robust code generation.

Important: You must output exclusively the finalized global consensus plan, prefixed strictly with "###". Do not expose your internal comparative analysis, resolution checklists, or intermediate reasoning steps in the final output. The generation of any concrete code implementations, conversational filler, or polite introductory words is strictly prohibited.

Figure 6: Prompt for Planning Consensus.

Architect Agent

Given a structured problem specification and a validated consensus plan, your objective is to perform a top-down architectural decomposition. You must synthesize a formal function dependency tree that delineates the modular skeleton of the target program without generating concrete implementations.

Problem Specification:
{problem_info}

Consensus Strategic Plan:
{consensus_plan}

Structural Constraints & Directives:
You must strictly construct the dependency tree according to the following architectural rules:
1. Entry Point Formulation
The root node of the dependency tree must be strictly designated as the primary execution interface, named `{entry}`.
2. Hierarchical Dependency Mapping
Each node must represent a distinct, user-defined sub-module. The `children` attribute of a node must encapsulate only the auxiliary functions it directly invokes. Leaf nodes represent terminal operations with empty dependency sets.
3. Declarative Abstraction
You must explicitly define only the function identifiers, structural signatures, and comprehensive docstrings detailing the expected behavioral logic. The generation of any concrete implementations or executable code is strictly prohibited.
4. Modularity Boundary
Maintain optimal structural granularity by constraining the entire tree to 2–5 essential functions. Avoid excessive fragmentation or over-engineering. Exclude all standard library routines and built-in functions from the graph.

Important: You must output exclusively a single valid JSON object representing the function dependency tree. Do not append any explanatory text, Markdown code block delimiters (e.g., ```json), conversational filler, or intermediate reasoning steps outside the JSON structure.

Figure 7: Prompt for Architect Agent.

Coder Agent

Given a complex competitive programming problem, generate functionally correct, highly optimized, and robust {language} code to solve the problem.

Contextual Information:
Problem Specification:
{problem_info}

Consensus Strategic Plan:
{consensus_plan}

Structural Skeleton (Function Dependency Tree):
```python
{tree_skeleton}

---
Important:
## Your response must contain only the {language} code to solve this problem. Do not add extra explanation or words.

Figure 8: Prompt for Coder Agent.

Tester Agent

Given a structured problem specification for a competitive programming task, your objective is to generate a robust and diverse set of test inputs.

# Problem Specification:
{problem_info}

# Test Generation Directives:
You must systematically construct the test suite by strictly adhering to the following criteria:
## 1. Input Schema Strictness
Each generated test case must perfectly align with the standard input (stdin) format delineated in the problem description. You must utilize the explicitly escaped `\n` character sequence to represent line breaks within each string-formatted test input.
## 2. Comprehensive Coverage Strategy
To effectively expose algorithmic vulnerabilities and evaluate structural integrity, the test suite must encompass a heterogeneous mix of scenarios, specifically targeting:
- Standard Operational Cases: Typical inputs reflecting the expected nominal distribution.
- Boundary & Edge Cases: Asymptotic limits, minimal/maximal constraints, and null/degenerate states.
- Stress-Testing Inputs: High-complexity configurations deliberately designed to challenge time complexities and trigger potential Time-Limit Exceeded (TLE) faults.

---
Important: You must output exclusively a single valid JSON array containing the generated string elements, formatted as `["input_1", "input_2", ...]`. Do not append any explanatory text, Markdown code block delimiters (e.g., ```json), conversational filler, or intermediate reasoning steps outside the JSON array.

Figure 9: Prompt for Tester Agent.
```

 # Problem:

```
def make_palindrome(string: str) -> str:
        Find the shortest palindrome that begins with a supplied string.
        Algorithm idea is simple:
        - Find the longest postfix of supplied string that is a palindrome.
        - Append to the end of the string reverse of a string prefix that comes before the palindromic suffix.
        >>> make_palindrome('')\n    ''
        >>> make_palindrome('cat')\n    'catac'
        >>> make_palindrome('cata')\n    'catac'
```

MapCoder:

```
def is_palindrome(string: str) -> bool:
    """ Test if given string is a palindrome
"""
    return string == string[::-1]

def make_palindrome(string: str) -> str:
        n = len(string)
        for i in range(n, 0, -1):
            suffix = string[i:]
            if suffix == suffix[::-1]:
                prefix = string[:i]
                return string + prefix[::-1]
```

DC-Coder:

```
def is_palindrome(string: str) -> bool:
    """ Test if given string is a palindrome """
    return string == string[::-1]

def make_palindrome(string: str) -> str:
    if not string:
        return ""
    n = len(string)
    max_k = 0
    for k in range(n, 0, -1):
        suffix = string[-k:]
        if is_palindrome(suffix):
            max_k = k
            break
    return string + string[:-max_k][::-1]
```

Figure 10: Example problem and solution generation using MapCoder, and DC-Coder prompts.