DebateCoder: Towards Collective Intelligence of LLMs via Test Case Driven LLM Debate for Code Generation

Jizheng Chen[1], **Kounianhua Du**[1], **Xinyi Dai**[2], **Weiming Zhang**[1],
Xihuai Wang[1], **Yasheng Wang**[2], **Ruiming Tang**[2], **Weinan Zhang**[1*], **Yong Yu**[1]
[1]Shanghai Jiao Tong University, [2]Huawei Noah's Ark Lab
Shanghai, China
{humihuadechengzhi,wnzhang}@sjtu.edu.cn

Abstract

With the impressive reasoning and text generation capabilities of large language models (LLMs), methods leveraging multiple LLMs to debate each other have garnered increasing attention. However, existing debate-based approaches remain limited in effectiveness in structured and detailed domains represented by code generation due to several reasons: 1) Reliance on different instances of the same LLM for debate, neglecting the potential benefits of integrating diverse models with varied internal knowledge for more comprehensive code generation, 2) under-utilization of test cases, and 3) reliance on third-party LLM moderators for result consolidation and decision-making, probably introducing hallucinations and judgment errors. To address these challenges, we propose DebateCoder to collect intelligence of LLMs via test case-driven debate for code generation. In DebateCoder, test cases serve as a medium for models to analyze code and identify bugs, while opposing models generate test cases to challenge each other's code during the debate process. These test cases, along with their execution results, are elaborately leveraged to refine and enhance the code through a novel contrastive analysis process. Furthermore, DebateCoder leverages test case outcomes to assess code quality and determine convergence criteria. Unlike previous approaches, DebateCoder emphasizes the collaborative improvement of both models through competitive debate and interactive analysis. Abundant experimental results on two datasets demonstrate the effectiveness of DebateCoder.

1 Introduction

Code generation is a critical yet challenging task that requires domain-specific knowledge and logical reasoning for continuous problem-solving. While the development of large language models (LLMs) has significantly advanced various natural language processing (NLP) tasks, raw NLP LLMs often encounter difficulties when addressing programming problems that necessitate a deep understanding of algorithms and precise code development. This challenge arises primarily due to the inherent differences between the open-ended nature of the NLP domain and the more structured requirements of code generation domain (Du et al., 2024). To overcome these limitations, existing approaches focus on adapting target LLMs to specialize in the code generation domain. Techniques such as fine-tuning large language models on extensive code corpora (Li et al., 2023b; Luo et al., 2023) or employing inference-time computational steps are commonly used, where the model iteratively refines its written program through repeated reasoning and search processes to identify more optimal solutions (Lu et al., 2023; Shinn et al., 2024). While these methods have notably enhanced the code generation capabilities of individual LLMs, they remain constrained by the limitations inherent to a single model, as its performance is bound by its internal coding capacity. This restricts the model's ability to engage in knowledge sharing or collaborative interactions with other models.

Recently, there has been a growing body of work leveraging multiple LLMs in a Mixture-of-Experts (MoE) framework, with debate-based methods gaining increasing attention (Liang et al., 2023; Long et al., 2024; Subramaniam et al., 2024). In a debate process, typically two or more models engage in a back-and-forth argumentative exchange, iteratively refining their responses, while a moderator evaluates the arguments and provides a consensus conclusion. While this approach helps overcome the limitations of a single LLM, when applied to the code generation domain, several problems remain unresolved: Firstly, debating models face challenges in accurately identifying flaws in each other's solutions, as two similar pieces of code may yield vastly different outputs, making it dif-

[*]Corresponding author.

ficult to pinpoint the underlying issues. Secondly, the reliance on third-party moderators often introduces errors and biases in judgment, which can lead to suboptimal final solutions. Thirdly, existing approaches fail to establish a clear connection between the generated code and the arguments put forth during the debate. These methods focus primarily on criticizing or debating the code itself, without incorporating execution feedback, thereby limiting opportunities for further solution refinement. Lastly, while test cases are a valuable tool for validating code correctness, current LLM debate frameworks do not effectively integrate them into the debate process.

To address the above challenges, in this paper, we propose DebateCoder, a test case-driven framework that incorporates a dual-model debate process for code generation. To effectively guide iterative and comprehensive thinking, DebateCoder refines the generated code through a competitive and collaborative debate process, where each model is engaged in a debate where they critique each other's code and then refine their own code through a comprehensive contrastive analysis process. To build the link between code scripts and debate arguments, during the debate, each model generates test cases to challenge the other model's solution. These test cases are then executed, with the resulting feedback used to drive the iterative refinement of the code. This contrasts with traditional methods that rely either on the internal feedback of a single model or a third-party moderator to make judgments, which can often be biased or error-prone.

To harness the collective intelligence of the two debate sides, DebateCoder fosters mutual improvement through a continuous co-evolutionary process. Rather than having each model merely "debate" or critique the other, both models actively participate in improving each other's code through targeted test case generation and execution. By doing so, DebateCoder effectively leverages the collective intelligence of both models, making the generated code more robust and accurate.

Furthermore, DebateCoder avoids the pitfalls of open-domain debate-based approaches by eliminating the need for a third-party moderator. The convergence criteria for the debate process are determined solely by the execution results of the test cases, ensuring that the final code solution is correct and optimized. This method significantly reduces the potential for judgment errors and hallucinations, which are common in existing LLM debate frameworks.

In summary, our main contributions can be summarized as follows:

- **DebateCoder framework.** We propose DebateCoder, a framework that integrates the collective intelligence of different debate sides within the code generation domain. To the best of our knowledge, DebateCoder is the first framework to combine the intelligence of both debate sides through an interactive debating process driven by test case generation.

- **Test cases as the medium for debate.** We leverage test case generation as the medium for the interactive debate, boosting the opponents to co-evolve through mutual test case generation and verification.

- **Adaptable convergence criteria for code generation.** Instead of relying on a third-party moderator, we use test case execution results to re-determine the convergence criteria for the debate process. Extensive Experimental results also validate the effectiveness of the DebateCoder framework.

2 Related Work

2.1 LLM for Code Generation

Large language models (LLMs) have gained widespread application in code generation domain due to their impressive abilities in both coding and reasoning. Current approaches can be generally categorized into three main groups: the first type involves fine-tuning pre-training LMs on extensive code corpora to enhance the models' understanding of code (Luo et al., 2023; Li et al., 2023b; Fried et al., 2022; Roziere et al., 2023; Bi et al., 2024; Hui et al., 2024). Due to high computational costs and scarcity of specialized training datasets, another line of work apply tuning-free methods like few-shot learning (Wang et al., 2022; Madaan et al., 2022) and retrieval-augmented generation (RAG) (Nashid et al., 2023; Du et al., 2024), which introduce domain knowledge into the model through external knowledge bases or prompts. A third line of work focuses on enhancing the model's internal reasoning process. Techniques such as Chain-of-Thought (CoT) (Yang et al., 2024b; Jiang et al., 2024; Li et al., 2023a), Tree-of-Thought (ToT) (Yao et al., 2024; La Rosa et al., 2024), and Monte Carlo Tree Search (MCTS) (Li et al., 2024; Zhang et al., 2023; Hu et al., 2024; Hao et al., 2023) are used to guide the model's problem-solving process. Other work prompts the model through a self-play process to reflect on previously generated contents to learn from itself (Haluptzok et al., 2022; Chen et al., 2023a; Lu et al., 2023; Chen et al., 2023b; Madaan et al., 2024; Shinn et al., 2024), or generates test cases as extra supervision signal (Chen et al., 2022; Huang et al., 2023). Despite achieving promising results, these models primarily focus on the reasoning process of a single model, overlooking the potential of leveraging the mutual intelligence of different models to further enhance performance in the code generation domain, which is the focus of this work.

2.2 Multi-Agent Debate for Reasoning

Multiple large models engaging in interactive debate can combine their respective arguments to push the system's performance limits (Lang et al., 2025). For instance, MAD (Liang et al., 2023) introduces a debate scenario with a moderator LLM guiding the models to provoke new thinking through a clash of viewpoints. MEP (Long et al., 2024) applies multiple models playing different roles and integrates their outputs, MapCoder (Islam et al., 2024) leverages multiple LLMs to emulate the developing cycle of human developers, while DebateGPT (Subramaniam et al., 2024) allows models to summarize each other's perspectives and perform data cleaning to ensure high-quality reasoning. DebateLLM (Du et al., 2023) iterates the models through the exchange of viewpoints. Existing work focuses on stubborn debate with a moderator in open domain, overlooking the collective improvement through a mutual promotion during the debate. In this paper, we focus on enhancing the debate process in code generation domain via comprehensive and collaborative model interaction driven by test cases.

3 Methodology

The framework of our proposed DebateCoder is illustrated in Figure 1. Our design has the following five key stages. Detailed prompts of each stage can be found in Figure 5 in the appendix part.

3.1 Zero-shot Solution Generation

In this stage, each model independently generates an initial solution to the given programming problem, which can be formulated as:

$$C_A = \mathcal{M}_{\mathcal{A}}(P_{zeroshot}(Q)),$$
$$C_B = \mathcal{M}_{\mathcal{B}}(P_{zeroshot}(Q)),$$

where Q is the problem description, $P_{zeroshot}$ denotes the zero-shot prompt construction process. $\mathcal{M}_{(\cdot)}$ represents LLM generation, and $C_{(\cdot)}$ is the generated code.

3.2 Self-Evolvement

The description section of a programming problem typically provides a certain number of sample test cases. These sample test cases may fail the zero-shot solutions, and thus can be leveraged for code refinement. In order to help both models identify and overcome shortcomings in their initial solutions, each model independently refines its zero-shot solution by leveraging these example test cases through self-evolvement.

Specifically, corresponding inputs and outputs are extracted from sample test cases. The initial solutions (C_A and C_B) generated in the zero-shot stage are executed against these test cases. Then the result pairs are collected for error analysis, where both debate models analyze the discrepancies between expected outputs and generated outputs for each failed test case.

Based on insights from the error analysis, each model generates an updated solution by incorporating feedback from the failed test cases into the solution refinement prompt. This results in the improved solutions, C_A^* and C_B^*.

3.3 Test Case Generation

The test case generation stage is a pivotal component of the DebateCoder framework, bridging the interactive refinement of solutions between the models. In this stage, each model generates test cases targeted at challenging the correctness and robustness of the opposing model's solution. Concretely, this stage consists of three sub-steps.

- **Problem analysis.** Each model is prompted to compare the opposing model's solution with its own refined solution from the self-evolvement stage, aiming to identify areas where the opposing solution might fail.
- **Test case construction.** Using insights from problem analysis, each model generates a test case specifically designed to highlight potential weaknesses or edge cases in the opposing model's solution, which can be formulated as:

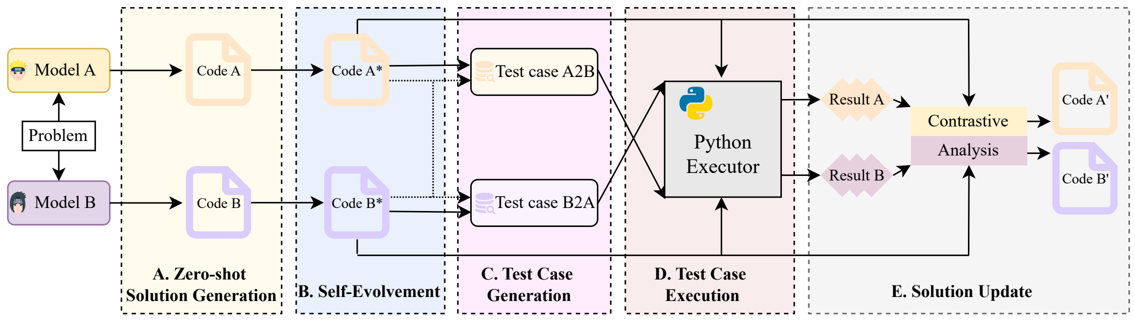


Figure 1: Framework overview. DebateCoder employs an iterative refinement to improve code generation by leveraging two models in a collaborative and competitive debate process via five key stages (marked as A to E).

$$T_{A2B} = \mathcal{M}_{\mathcal{A}}(P_{testcase}(Q, C_B^*, I_B)),$$
$$T_{B2A} = \mathcal{M}_{\mathcal{B}}(P_{testcase}(Q, C_A^*, I_A)),$$

where Q is the problem description, C_A^* and C_B^* are the refined solutions from the self-evolvement stage, $P_{testcase}$ represents the prompt construction for test case generation, $I_{(\cdot)}$ denotes insight from problem analysis process, and T_{A2B} and T_{B2A} are the test cases generated by Models A and B to challenge each other.

The generated test case is capsulated in a JSON format dictionary, with "Input" as the key and the test case string as the value.

- **Test case validation.** To ensure that generated test cases are logically consistent with the problem constraints and can be completely executed on the challenger model's solution, they are verified for validity. Specifically, we check the output information of T_{A2B} on C_A^* and T_{B2A} on C_B^*. If an error is reported, a correction process is conducted to regenerate the test cases.

By encouraging models to challenge each other via executable test cases, DebateCoder drives iterative refinement with actionable feedback.

3.4 Test Case Execution

To ensure that during the debate the models efficiently leverage previously generated test cases and achieve incremental performance improvements, in every iteration epoch, the test case from the opposing debate model along with execution results on the challenger's solution is stored in a test case pool. The whole process can be formulated as:

$$Res_A = Exe(C_B^*, Input_{B2A}),$$
$$Res_B = Exe(C_A^*, Input_{A2B}),$$
$$pool_A \leftarrow \langle Input_{B2A}, Res_A \rangle,$$
$$pool_B \leftarrow \langle Input_{A2B}, Res_B \rangle,$$

where $pool_{(\cdot)}$ is the memory pool to store test cases raised by opposing debate model, $Input_{(\cdot)}$ and $Res_{(\cdot)}$ denote test case input and corresponding execution results. Exe is the Python executor.

3.5 Solution Update

In this stage, each model integrates insights from test case execution to improve its solution, leading to progressively more robust generation results, which can be decomposed into two sub-stages:

- **Error analysis.** Each model examines the discrepancies between the execution results of both models' solutions on the opposing model's test cases. This process is done using a novel *contrastive analysis* approach, in which each model is additionally given the opposing model's solution and execution results. Then, the model aims to comprehensively compare the execution results and logic differences between the two pieces of code, and perform error analysis based on this. Through this step, our framework not only identifies weaknesses but also learns from the strengths in the counterpart's code and generates targeted test cases.

- **Solution refinement.** Using the error analysis results, each model generates specific feedback on how its solution can be improved. Then, by incorporating the feedback derived from the test cases and execution results, each model updates its solution. The refined solutions C'_A and C'_B are expected to correct the identified issues, enhancing robustness against edge cases introduced by the opposing model. The process can be mathematically expressed as:

$$C'_A = \mathcal{M}_{\mathcal{A}}(P_{refine}(Q, T_{B2A}, Res_A, I_A)),$$
$$C'_B = \mathcal{M}_{\mathcal{B}}(P_{refine}(Q, T_{A2B}, Res_B, I_B)),$$

where P_{refine} represents the prompt for solution refinement, T_{B2A} and T_{A2B} are test cases generated by the opposing models, Res_A and Res_B are execution results, and C'_A and C'_B denote the updated solutions.

3.6 DebateCoder: Convergence Criteria

Different from previous work where a moderator LLM is used to determine the termination of debate, we split dataset test cases into public and private test cases by half, and leverage the following convergence criteria to determine whether to stop at the end of each debate epoch:

- **Maximum iteration limit.** The process halts if the number of iterations exceeds a predefined maximum threshold.
- **Complete validation pass.** The process stops immediately if both solutions pass all test cases in the public test case set, demonstrating robustness and correctness.
- **Public set performance.** If performance on the public test case set improves, the refined solutions are retained. If performance deteriorates, an early stopping mechanism is triggered where the refinement process is stopped when the solutions show repeated performance degradation on the public set for a predefined number of epochs.

4 Experiment

In this section, we conduct a series of experiments to answer the following research questions (RQs):

RQ1 How does our proposed DebateCoder perform against the baselines?

RQ2 What is the trend of the model's performance with a different number of debate epochs?

RQ3 Is DebateCoder compatible with open-source large language models?

RQ4 Do generated test cases and convergence criteria bring performance gain, realizing the collective intelligence of debate sides?

RQ5 What is the concrete difference between typical debate-based methods and DebateCoder in code generation domain?

4.1 Setup

4.1.1 Datasets

We evaluate DebateCoder and the competing methods on the popular benchmark datasets APPS (Hendrycks et al., 2021) and CodeContest (Li et al., 2022). APPS dataset has three levels of difficulties, namely introductory, interview, and competition, and the first 100 problems in each level are used for evaluation. For CodeContest dataset, we split the test problems into basic and advanced levels according to their difficulty tags, getting over 100 basic problems and around 60 advanced problems. Following previous works (Austin et al., 2021; Chen et al., 2021; Dong et al., 2023), *pass rate* and *pass@1* are used as the evaluation metric for code generation correctness, where *pass rate* represents the average percentage of private test cases that the generated programs pass across all problems, while *pass@1* indicates the percentage of problems in which the generated programs successfully pass all private test cases.

4.1.2 Baselines

We compare with a series of competitive methods to validate the effectiveness of our proposed DebateCoder, including methods that involve self-refinement and iterative reasoning: self-play (Madaan et al., 2022), Reflexion (Shinn et al., 2024), CoT (Yang et al., 2024b), and two debate-based methods: MEP (Long et al., 2024), MAD (Liang et al., 2023) along with code generation methods CodeT (Chen et al., 2022), LDB (Zhong et al., 2024) and AgentCoder (Huang et al., 2023).

4.1.3 Implementation

We select Claude-3.5-sonnet and GPT-4o-mini as the two opposing backbones of debate. For the debate-based methods, MEP, MAD, and DebateCoder, the number of maximum debate epochs is set to 10. We adjust the prompt for MEP and MAD to fit the code generation task. We split half of the problem test cases as the public test cases for convergence criteria, following the settings in previous work (Li et al., 2024; Chen et al., 2022), and use the other half as a private set.

4.2 Overall Performance (RQ1)

In this section, we compare our proposed DebateCoder with various code generation baseline models. Their performance is listed in Table 1. Our code is available[1].

From the results, one can observe that: (1) DebateCoder outperforms all the baseline models, demonstrating the effectiveness of our proposed debate framework that leverages test case generation

[1]https://github.com/Otsuts/DebateCoder

Table 1: Major results. For each backbone, the best result(s) are marked in bold, and the second best result(s) are underlined. Each method is run over three times to get the best result for a fair comparison.

Arch	Method	APPS						CodeContest			
		Pass Rate			Pass@1			Pass Rate		Pass@1	
		Intro.	Inter.	Comp.	Intro.	Inter.	Comp.	Basic	Advance	Basic	Advanced
claude-3.5-sonnet	zero-shot	0.6456	0.6297	0.4816	0.4500	0.4200	0.3300	0.4990	0.4664	0.3750	0.2581
	self-play	0.7756	0.7463	0.5316	**0.5700**	0.5400	0.3500	0.5085	0.4871	0.3882	0.3871
	Reflexion	0.7719	0.7417	0.5403	0.5100	0.5300	0.3500	0.5396	0.5314	0.3975	0.3838
	MAD	0.7487	0.7099	0.4933	0.5000	0.4500	0.3600	0.5186	0.5078	0.3906	0.3188
	MEP	0.7568	0.7358	0.4983	0.5200	0.4500	0.3400	0.5107	0.5056	0.3856	0.3281
	zero-shot CoT	0.6710	0.6363	0.5025	0.4900	0.4800	0.3700	0.5086	0.4945	0.3862	0.3614
	CodeT	0.6777	0.6584	0.5000	0.4600	0.4500	0.3200	0.5587	0.5621	0.4521	0.4186
	LDB	0.6815	0.6242	0.5267	0.5400	0.5500	0.3900	0.6017	0.5141	0.5323	0.4419
	AgentCoder	0.7560	0.7258	0.5283	0.5100	0.4800	0.3300	0.5500	0.5603	0.4052	0.3593
	DebateCoder	**0.7932**	**0.7736**	**0.5681**	**0.5700**	**0.5500**	**0.4100**	**0.6174**	**0.6003**	**0.5620**	**0.4688**
gpt-4o-mini	zero-shot	0.5773	0.5780	0.3467	0.2900	0.2900	0.1600	0.3028	0.2814	0.1830	0.1250
	self-play	0.6419	0.6882	0.4006	0.3700	**0.4300**	**0.2000**	0.3850	0.3511	0.2434	0.1452
	Reflexion	0.6381	0.6352	0.3833	0.4000	0.4100	0.1800	0.3381	0.2871	0.1908	0.1625
	MAD	0.5591	0.6479	0.3488	0.3800	0.3200	0.1900	0.3366	0.2942	0.2157	0.1406
	MEP	0.6181	0.6080	0.3600	0.3900	0.3200	0.1600	0.3499	0.2879	0.2449	0.1450
	zero-shot CoT	0.5580	0.5360	0.3387	0.3900	0.3400	0.1500	0.3437	0.3149	0.2222	0.1416
	CodeT	0.6187	0.5798	0.3933	**0.4500**	0.3300	0.1800	0.4049	0.3671	0.3168	**0.2500**
	LDB	0.6011	0.5765	0.3800	0.4200	0.4100	0.1600	0.3065	0.3352	0.2093	0.1628
	AgentCoder	0.6368	0.5805	0.3566	0.3800	0.3300	0.1600	0.3833	0.2926	0.2007	0.1875
	DebateCoder	**0.7236**	**0.6980**	**0.4457**	**0.4500**	**0.4300**	**0.2000**	**0.4556**	**0.3920**	**0.3856**	**0.2500**

for flaw detection and code refinement. (2) Among the baseline models, self-play, CodeT and LDB achieve the best performance in most settings, indicating that iterative self-reflection driven by generated test cases provides valuable feedback that significantly boosts model performance. However, our model incorporates a process where the two models engage in mutual debate and co-improvement based on the test cases, leading to better results. (3) Our method also surpasses the debate-based method MEP and MAD in open domains, suggesting that DebateCoder is better suited for the code generation domain.

4.3 Performance Change Over Debate Epochs (RQ2)

In this section, we study the performance gain over every epoch of the whole debate process on the APPS dataset. We record the average debate epochs before the debate terminates in Table 2.

Table 2: Average debate epochs.

	APPS(Intro.)	APPS(Inter.)	APPS(Comp.)
# Average Epochs	2.01	2.04	1.54

Based on this, we plot the pass rate change curves of both debate models across each epoch in Figure 2. Since the performance of both sides converges in the later rounds, we only recorded the performance trends for the first 6 epochs. Due to space limitations, the specific data is presented in

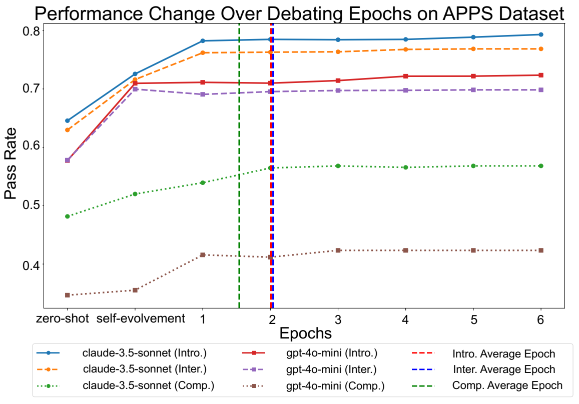


Figure 2: Performance change over debate epochs.

Table 4 in the appendix part.

From the table and figure, it can be seen that DebateCoder demonstrates good convergence, with performance stabilizing around the first two debate epochs. Furthermore, one can observe that through self-evolvement, the debate models can learn from the visible high-quality sample test cases, resulting in stable performance improvements. Building on this, our model can continue to achieve steady performance growth through test case generation, even without additional visible test cases.

4.4 Compatibility Study (RQ3)

In this section, we implement DebateCoder based on two open source LLMs, Qwen-32b-Instruct (Yang et al., 2024a) and Yi-34b-Chat (Young et al., 2024), and compare them with baselines mentioned in Section 4.1.2 on APPS dataset.

Table 3: Performance of open-source models on APPS. For each backbone, the best result(s) are marked in bold, and the second best result(s) are underlined.

Arch	Method	APPS Pass Rate Intro.	APPS Pass Rate Inter.	APPS Pass Rate Comp.	APPS Pass@1 Intro.	APPS Pass@1 Inter.	APPS Pass@1 Comp.
Qwen-32b-Instruct	zero-shot	0.6295	0.5348	0.3317	0.4300	0.2300	0.1600
	self-play	0.7677	0.7134	0.3633	0.4800	0.4300	0.1800
	Reflexion	0.5937	0.5600	0.3750	0.4700	0.3600	0.2100
	MAD	0.6337	0.5506	0.3535	0.4400	0.2500	0.1600
	MEP	0.7113	0.5624	0.3788	0.4400	0.2900	0.1800
	CodeT	0.6336	0.6089	0.3750	0.4500	0.3500	0.1900
	LDB	0.6329	0.5831	0.3300	0.4500	0.4700	0.2000
	AgentCoder	0.6553	0.6186	0.3850	0.4300	0.3500	0.2200
	DebateCoder	**0.7709**	**0.7176**	**0.3983**	**0.4900**	**0.4900**	**0.2300**
Yi-34b-Chat	zero-shot	0.3968	0.3813	0.2650	0.2400	0.1800	0.1200
	self-play	0.5309	0.5710	0.3000	0.3400	0.3300	0.1600
	Reflexion	0.4868	0.4863	0.3022	0.3000	0.2700	0.1600
	MAD	0.4340	0.5129	0.2245	0.2700	0.2200	0.0900
	MEP	0.4632	0.5264	0.2837	0.2900	0.2700	0.1400
	CodeT	0.4270	0.5121	0.2950	0.2500	0.2600	0.1500
	LDB	0.4464	0.4352	0.2283	0.2700	0.2600	0.1200
	AgentCoder	0.5177	0.5080	0.2483	0.3500	0.3000	0.1300
	DebateCoder	**0.6689**	**0.6750**	**0.3216**	**0.4400**	**0.3400**	**0.1800**

From the experimental results, it can be seen that DebateCoder achieves the best performance, surpassing all baseline models. This demonstrates that the performance of our model is independent of the base model and exhibits robustness, verifying the effectiveness of DebateCoder.

4.5 Ablation Study (RQ4)

In this section, we conduct ablation experiments to evaluate the effectiveness of each component of DebateCoder. We modify the proposed framework into a series of variants, as outlined below:

- **Best-of-N (①).** The backbone large language model is prompted to perform N iterations of zero-shot code generation, and the best result will be selected as the final score.
- **DebateCoder-Same (②).** The same backbone model is used for both debate sides, with all other settings remaining unchanged.
- **DebateCoder-Moderator (③).** Similar to MAD (Liang et al., 2023), a third-party moderator is introduced to assess the quality of responses from both debate models in each round and provide the final refined code.
- **DebateCoder-Explain (④).** The test case generation component from the original framework is replaced with direct analysis and refinement of the generated code.

We compare the variants with our proposed DebateCoder mentioned in Section 3, denoted here as DebateCoder-Testcase (⑤). Results are presented in Figure 3. The following conclusion can be drawn from the results:

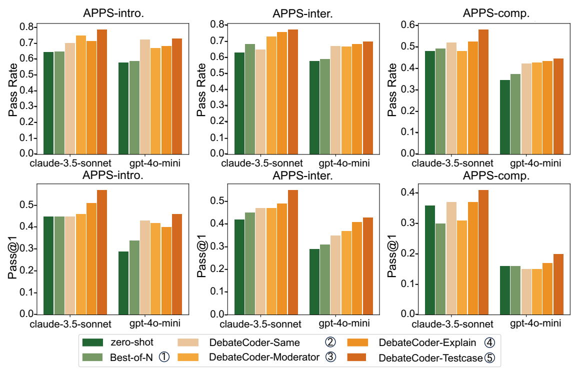


Figure 3: Ablation study: performance of variants of DebateCoder.

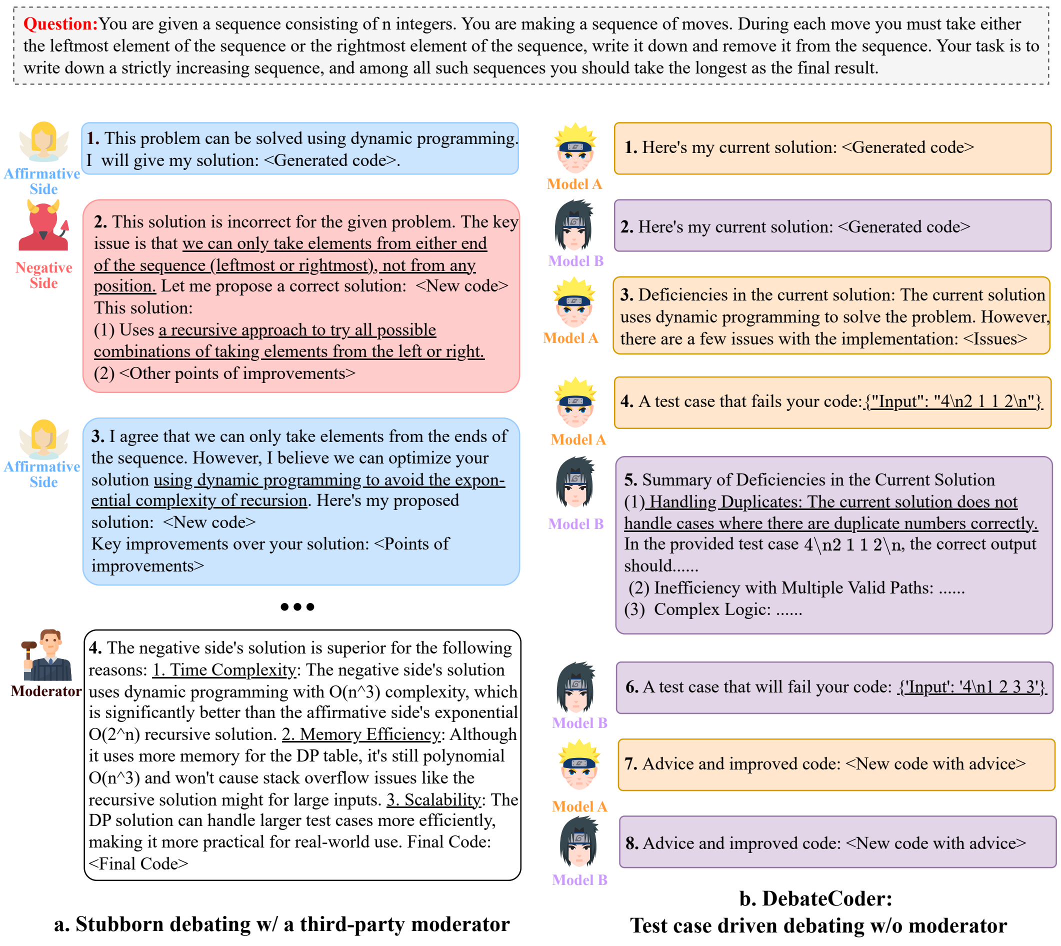
Impact of DebateCoder's debate paradigm (①,② VS ⑤). Although the claude-3.5-sonnet model outperforms gpt-4o-mini in code capabilities, when both debate sides are switched to claude-3.5-sonnet, the performance of DebateCoder actually decreases. This suggests that different models possess distinct internal knowledge of code-related problems. DebateCoder effectively integrates and promotes the knowledge of these models, achieving the effect of collective intelligence. Similarly, our approach also outperforms multiple generations that rely on previously generated outputs. This demonstrates that the framework is well-suited for code generation tasks.

Impact of test cases (④ VS ⑤). One can observe that the use of test cases further improves the model's performance, indicating that the approach of DebateCoder, which uses generated test cases to identify issues in the opponent's code and refines the code based on feedback from test case execution, is effective.

Impact of the third-party moderator (③ VS ⑤). Existing debate-based methods mainly rely on a third-party moderator model to evaluate responses and summarize issues from both sides. However, as shown in the figure, DebateCoder achieves better results by using test case execution results to determine convergence conditions and control the number of debate rounds.

4.6 Case Study (RQ5)

Existing debate-based methods are primarily applied in open-domain knowledge question-answering tasks, where the debate models identify overlooked or erroneous flaws in previously generated results through rebuttal processes, and a third-


Question: You are given a sequence consisting of n integers. You are making a sequence of moves. During each move you must take either the leftmost element of the sequence or the rightmost element of the sequence, write it down and remove it from the sequence. Your task is to write down a strictly increasing sequence, and among all such sequences you should take the longest as the final result.

Affirmative Side: **1.** This problem can be solved using dynamic programming. I will give my solution: <Generated code>.

Negative Side: **2.** This solution is incorrect for the given problem. The key issue is that we can only take elements from either end of the sequence (leftmost or rightmost), not from any position. Let me propose a correct solution: <New code> This solution:
(1) Uses a recursive approach to try all possible combinations of taking elements from the left or right.
(2) <Other points of improvements>

Affirmative Side: **3.** I agree that we can only take elements from the ends of the sequence. However, I believe we can optimize your solution using dynamic programming to avoid the exponential complexity of recursion. Here's my proposed solution: <New code>
Key improvements over your solution: <Points of improvements>

•••

Moderator: **4.** The negative side's solution is superior for the following reasons: 1. Time Complexity: The negative side's solution uses dynamic programming with O(n^3) complexity, which is significantly better than the affirmative side's exponential O(2^n) recursive solution. 2. Memory Efficiency: Although it uses more memory for the DP table, it's still polynomial O(n^3) and won't cause stack overflow issues like the recursive solution might for large inputs. 3. Scalability: The DP solution can handle larger test cases more efficiently, making it more practical for real-world use. Final Code: <Final Code>

a. Stubborn debating w/ a third-party moderator

Model A: **1.** Here's my current solution: <Generated code>

Model B: **2.** Here's my current solution: <Generated code>

Model A: **3.** Deficiencies in the current solution: The current solution uses dynamic programming to solve the problem. However, there are a few issues with the implementation: <Issues>

Model A: **4.** A test case that fails your code: {"Input": "4\n2 1 1 2\n"}

Model B: **5.** Summary of Deficiencies in the Current Solution
(1) Handling Duplicates: The current solution does not handle cases where there are duplicate numbers correctly. In the provided test case 4\n2 1 1 2\n, the correct output should......
(2) Inefficiency with Multiple Valid Paths: ......
(3) Complex Logic: ......

Model B: **6.** A test case that will fail your code: {'Input': '4\n1 2 3 3'}

Model A: **7.** Advice and improved code: <New code with advice>

Model B: **8.** Advice and improved code: <New code with advice>

b. DebateCoder: Test case driven debating w/o moderator

Figure 4: Case study of MAD and DebateCoder on a dynamic programming problem of APPS. Key points of model response are underlined.

party moderation model is introduced to evaluate the outcomes. This paradigm, however, has the following issues when adapted to code generation domain: (1) a lack of effective mechanisms to determine the correctness of code and obtain execution results, making it difficult to accurately identify flaws in the opponent's code; (2) the third-party moderator model suffers from judgment biases and hallucination problems. These issues result in poor performance when such methods are adapted to the code generation domain.

In this section, to reveal the core difference between the typical debate-based method and DebateCoder, we use a case study comparing the detailed behavior of the open-domain debate-based method MAD and DebateCoder on a specific programming problem. The problem descriptions and model responses are summarized in Figure 4.

It is evident that in MAD, the debate sides identified issues with the dynamic programming implementation (Response 2) and the algorithm's time complexity (Response 3). However, they failed to detect the primary error in the code (incorrect input data handling), leading to an incorrect result. In contrast, DebateCoder identified the error in input data handling (Response 5) through the execution results of generated test cases (Response 4) and constructed targeted test cases (Response 6). The final modified result successfully passed most data points. This validates the effectiveness of DebateCoder in the code generation domain.

5 Conclusion

In this paper we propose DebateCoder, a test case-driven framework for code generation that employs dual-model debate for code generation. By utilizing test cases for models to critique and refine each other's code, DebateCoder integrates the collective intelligence of different debate sides and addresses the limitations of typical debate-based methods, ensuring both correctness and optimization of generated code. Experiment results highlight the potential of DebateCoder to improve the reliability and efficiency of code generation tasks.

Acknowledgements

The Shanghai Jiao Tong University team is partially supported by National Key R&D Program of China (2022ZD0114804), Shanghai Municipal Science and Technology Major Project (2021SHZDZX0102) and National Natural Science Foundation of China (62322603, 62177033).

Limitations

While DebateCoder introduces a novel test case-driven approach for large language model (LLM) debate in code generation, certain limitations remain that could be addressed in future research:

Restricted to Two-Model Debate Our current framework focuses on the debate between two LLMs, leveraging their complementary knowledge to refine code generation. However, extending the debate to a multi-agent setting, where more than two models with varying architectures or training paradigms engage in a structured discussion, could further enhance the robustness and coverage of edge cases.

Debating Granularity: Code-Level vs. Thought-Level DebateCoder emphasizes debating over entire generated code snippets, with test cases serving as the primary medium for critique. While effective, this approach does not explicitly model a thought-level debate process, where models critically analyze intermediate reasoning steps before arriving at a final code solution. Introducing step-wise thought validation and thought-level debate could provide finer-grained feedback and improve overall reasoning capabilities.

Lack of Adaptivity Through Learning DebateCoder currently operates purely during inference time, without mechanisms for parameter updates. As test case results are non-differentiable, the framework does not support end-to-end gradient-based learning. Future extensions could investigate how to incorporate reinforcement learning techniques to enable adaptive improvement over time by using test outcomes as reward signals.

These limitations highlight areas where DebateCoder can evolve. We believe that addressing them could further enhance the framework's adaptability and effectiveness across various structured reasoning tasks.

References

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

Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, et al. 2024. Deepseek llm: Scaling open-source language models with longtermism. *arXiv preprint arXiv:2401.02954*.

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*.

Kai Chen, Chunwei Wang, Kuo Yang, Jianhua Han, Lanqing Hong, Fei Mi, Hang Xu, Zhengying Liu, Wenyong Huang, Zhenguo Li, et al. 2023a. Gaining wisdom from setbacks: Aligning large language models via mistake analysis. *arXiv preprint arXiv:2310.10477*.

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

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

Yihong Dong, Jiazheng Ding, Xue Jiang, Ge Li, Zhuo Li, and Zhi Jin. 2023. Codescore: Evaluating code generation by learning code execution. *arXiv preprint arXiv:2301.09043*.

Kounianhua Du, Renting Rui, Huacan Chai, Lingyue Fu, Wei Xia, Yasheng Wang, Ruiming Tang, Yong Yu, and Weinan Zhang. 2024. Codegrag: Extracting composed syntax graphs for retrieval augmented cross-lingual code generation. *arXiv preprint arXiv:2405.02355*.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B Tenenbaum, and Igor Mordatch. 2023. Improving factuality and reasoning in language models through multiagent debate. *arXiv preprint arXiv:2305.14325*.

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*.

Patrick Haluptzok, Matthew Bowers, and Adam Tauman Kalai. 2022. Language models can teach themselves to program better. *arXiv preprint arXiv:2207.14502*.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Jiahua Hong, Zhen Wang, Daisy Zhe Wang, and Zhiting Hu. 2023. Reasoning with language model is planning with world model. *arXiv preprint arXiv:2305.14992*.

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

Zhiyuan Hu, Chumin Liu, Xidong Feng, Yilun Zhao, See-Kiong Ng, Anh Tuan Luu, Junxian He, Pang Wei Koh, and Bryan Hooi. 2024. Uncertainty of thoughts: Uncertainty-aware planning enhances information seeking in large language models. *arXiv preprint arXiv:2402.03271*.

Dong Huang, Qingwen Bu, Jie M Zhang, Michael Luck, 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, et al. 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. *arXiv preprint arXiv:2405.11403*.

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.

Ricardo La Rosa, Corey Hulse, and Bangdi Liu. 2024. Can github issues be solved with tree of thoughts? *arXiv preprint arXiv:2405.13057*.

Hao Lang, Fei Huang, and Yongbin Li. 2025. Debate helps weak-to-strong generalization. *Preprint*, arXiv:2501.13124.

Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2023a. Structured chain-of-thought prompting for code generation. *ACM Transactions on Software Engineering and Methodology*.

Qingyao Li, Wei Xia, Kounianhua Du, Xinyi Dai, Ruiming Tang, Yasheng Wang, Yong Yu, and Weinan Zhang. 2024. Rethinkmcts: Refining erroneous thoughts in monte carlo tree search for code generation. *arXiv preprint arXiv:2409.09584*.

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

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Tian Liang, Zhiwei He, Wenxiang Jiao, Xing Wang, Yan Wang, Rui Wang, Yujiu Yang, Shuming Shi, and Zhaopeng Tu. 2023. Encouraging divergent thinking in large language models through multi-agent debate. *arXiv preprint arXiv:2305.19118*.

Do Xuan Long, Duong Ngoc Yen, Anh Tuan Luu, Kenji Kawaguchi, Min-Yen Kan, and Nancy F Chen. 2024. Multi-expert prompting improves reliability, safety, and usefulness of large language models. *arXiv preprint arXiv:2411.00492*.

Jianqiao Lu, Wanjun Zhong, Wenyong Huang, Yufei Wang, Fei Mi, Baojun Wang, Weichao Wang, Lifeng Shang, and Qun Liu. 2023. Self: Language-driven self-evolution for large language model. *arXiv preprint arXiv:2310.00533*.

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*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2024. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36.

Aman Madaan, Shuyan Zhou, Uri Alon, Yiming Yang, and Graham Neubig. 2022. Language models of code are few-shot commonsense learners. *arXiv preprint arXiv:2210.07128*.

Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-based prompt selection for code-related few-shot learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2450–2462. IEEE.

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

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36.

Vighnesh Subramaniam, Antonio Torralba, and Shuang Li. 2024. Debategpt: Fine-tuning large language models with multi-agent debate supervision.

Xingyao Wang, Sha Li, and Heng Ji. 2022. Code4struct: Code generation for few-shot event structure prediction. *arXiv preprint arXiv:2210.12810*.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. 2024a. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*.

Guang Yang, Yu Zhou, Xiang Chen, Xiangyu Zhang, Terry Yue Zhuo, and Taolue Chen. 2024b. Chain-of-thought in neural code generation: From and for lightweight language models. *IEEE Transactions on Software Engineering*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2024. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36.

Alex Young, Bei Chen, Chao Li, Chengen Huang, Ge Zhang, Guanwei Zhang, Heng Li, Jiangcheng Zhu, Jianqun Chen, Jing Chang, et al. 2024. Yi: Open foundation models by 01. ai. *arXiv preprint arXiv:2403.04652*.

Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B Tenenbaum, and Chuang Gan. 2023. Planning with large language models for code generation. *arXiv preprint arXiv:2303.05510*.

Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Debug like a human: A large language model debugger via verifying runtime execution step-by-step. *arXiv preprint arXiv:2402.16906*.

A Appendix

A.1 Performance Change Over Debate Epochs

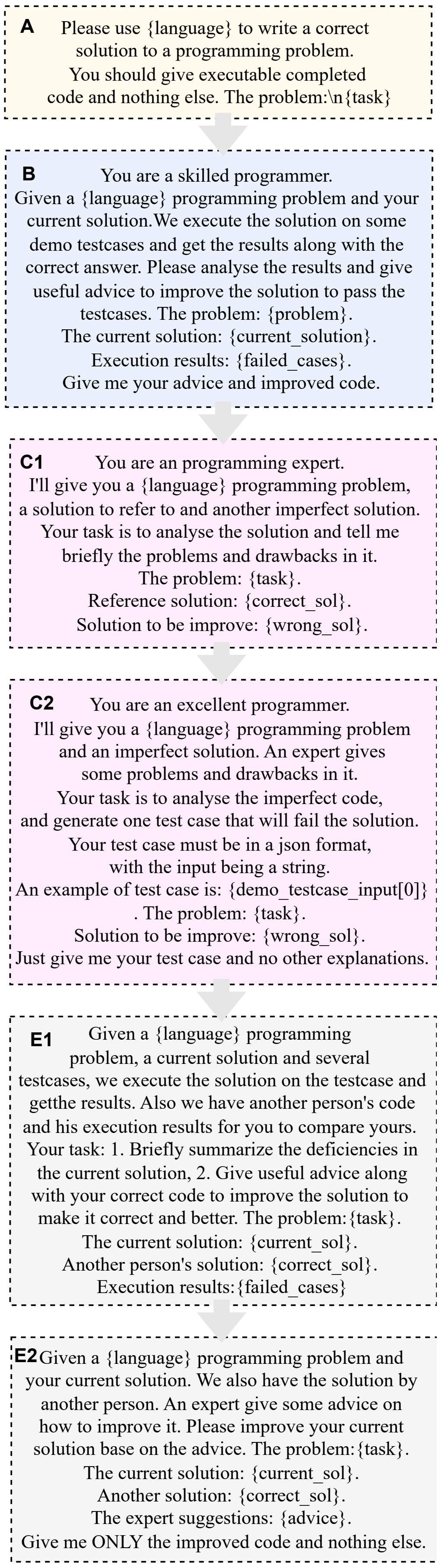
In Section 4.3 we study performance gain over every epoch of the whole debate process. We record the table corresponding to Figure 2 in Table 4.

Table 4: Performance change over debate epochs. 'self-evolve' represents the refined program after self-evolvement stage in Section 3.2.

Arch	Debate-epoch	APPS(Intro.)	APPS(Inter.)	APPS(Comp.)
claude-3.5-sonnet	zero-shot	0.6456	0.6297	0.4816
	self-evolve	0.7257	0.7161	0.5200
	1	0.7824	0.7620	0.5393
	2	0.7849	0.7630	0.5646
	3	0.7843	0.7636	0.5681
	4	0.7849	0.7677	0.5656
	5	0.7887	0.7686	0.5681
	6	0.7932	0.7686	0.5681
gpt-4o-mini	zero-shot	0.5773	0.5780	0.3467
	self-evolve	0.6251	0.6240	0.3552
	1	0.7113	0.6907	0.4157
	2	0.7099	0.6954	0.4118
	3	0.7143	0.6973	0.4234
	4	0.7218	0.6976	0.4234
	5	0.7219	0.6984	0.4234
	6	0.7236	0.6984	0.4234

A.2 Detailed prompt of each stage

In Section 3, we introduced the overall framework of DebateCoder. We list the detailed prompts of each stage corresponding to Figure 1 in Figure 5.



A Please use {language} to write a correct solution to a programming problem. You should give executable completed code and nothing else. The problem:\n{task}

B You are a skilled programmer. Given a {language} programming problem and your current solution.We execute the solution on some demo testcases and get the results along with the correct answer. Please analyse the results and give useful advice to improve the solution to pass the testcases. The problem: {problem}. The current solution: {current_solution}. Execution results: {failed_cases}. Give me your advice and improved code.

C1 You are an programming expert. I'll give you a {language} programming problem, a solution to refer to and another imperfect solution. Your task is to analyse the solution and tell me briefly the problems and drawbacks in it. The problem: {task}. Reference solution: {correct_sol}. Solution to be improve: {wrong_sol}.

C2 You are an excellent programmer. I'll give you a {language} programming problem and an imperfect solution. An expert gives some problems and drawbacks in it. Your task is to analyse the imperfect code, and generate one test case that will fail the solution. Your test case must be in a json format, with the input being a string. An example of test case is: {demo_testcase_input[0]} . The problem: {task}. Solution to be improve: {wrong_sol}. Just give me your test case and no other explanations.

E1 Given a {language} programming problem, a current solution and several testcases, we execute the solution on the testcase and getthe results. Also we have another person's code and his execution results for you to compare yours. Your task: 1. Briefly summarize the deficiencies in the current solution, 2. Give useful advice along with your correct code to improve the solution to make it correct and better. The problem:{task}. The current solution: {current_sol}. Another person's solution: {correct_sol}. Execution results:{failed_cases}

E2 Given a {language} programming problem and your current solution. We also have the solution by another person. An expert give some advice on how to improve it. Please improve your current solution base on the advice. The problem:{task}. The current solution: {current_sol}. Another solution: {correct_sol}. The expert suggestions: {advice}. Give me ONLY the improved code and nothing else.

Figure 5: Prompts of each stage in DebateCoder.