MACS CODER: A Multi-Agent Coding Framework for Small LMs — From Fast Thinking to Deep Planning

Anonymous ACL submission

Abstract

Current multi-agent coding frameworks are often resource-intensive, employing a one-size-fits-all strategy that lacks efficiency. We propose **MACS-Coder** (Multi-Agent Adaptive Coding Structure), a dual-process framework inspired by human cognition. It adaptively switches between a **Fast Thinking System** for rapid, low-cost generation and a **Deep Planning System**—comprising specialized planning, templating, and debugging agents—for complex tasks. This architecture enables compact open-source models to achieve performance comparable to elite proprietary systems with significantly lower energy consumption. Extensive evaluations show that MACS-Coder achieves new SOTA results among open-source models: using a `gpt-oss-20B` backbone, it attains **99.4%** on HumanEval, **93.2%** on MBPP, and **83.2%** on LiveCodeBench V5, consistently outperforming prior methods like CodeSIM and MapCoder. Notably, our 20B-parameter framework achieves performance competitive with top-tier models such as o4-mini and Gemini 2.5 Pro, effectively narrowing the gap between small open-source and large closed-source systems. We will open-source our framework and evaluation suite available here.

1 Introduction

In recent years, the rapid advancement of Large Language Models (LLMs) has fundamentally transformed AI-assisted programming. These models have revolutionized how developers generate, reason about, and debug code, significantly lowering the barrier to entry for software creation and enabling tasks ranging from routine maintenance to building complex applications from scratch.

While flagship proprietary models like OpenAI's GPT-5 line (OpenAI, 2025c) and Anthropic's CLAUDE-4 (Anthropic, 2025) continue to define the state of the art, a rapidly expanding ecosystem of open-source families (e.g., Meta's LLAMA-4 (Meta AI, 2025), Alibaba's QWEN-3 (Yang et al., 2025)) is narrowing the performance gap. However, two practical limitations persist. First, top-tier proprietary models rely on immense parameter counts and compute budgets, restricting access for many researchers and production systems. Second, while modern multi-agent frameworks such as MapCoder (Islam et al., 2024) and CodeSIM (Islam et al., 2025) improve robustness through decomposed reasoning stages, they typically adopt a static, **"one-size-fits-all"** strategy. These systems invoke a complete, resource-intensive stack for every problem instance, regardless of difficulty. Despite these advances, current systems lack the ability to adapt reasoning depth at the instance level, resulting in systematic inefficiency.

Small Language Models (SLMs) (parameters $\lesssim$ 20B) offer a cost-effective alternative but often lag in complex reasoning. This raises a central research question: *How can we leverage a specialized agent framework to enable SLMs to bridge the capability gap with large models? In particular, can a structured process architecture simultaneously reduce computational costs and enhance system stability?*

To address this, we propose **MACS-Coder**, a framework inspired by the dual-process theory of human cognition. MACS-Coder integrates three core innovations: (1) a *Fast-and-Deep Planning* architecture that dynamically selects between a low-latency fast path for easy instances and a deliberative deep-planning pipeline for harder ones; (2) *structured generation* via an STD-IO Tool to enhance output stability; and (3) a *fine-grained debugging* mechanism that precisely isolates and repairs errors. The architecture is illustrated in Figure 1, with the deep reasoning workflow detailed in Figure 2.

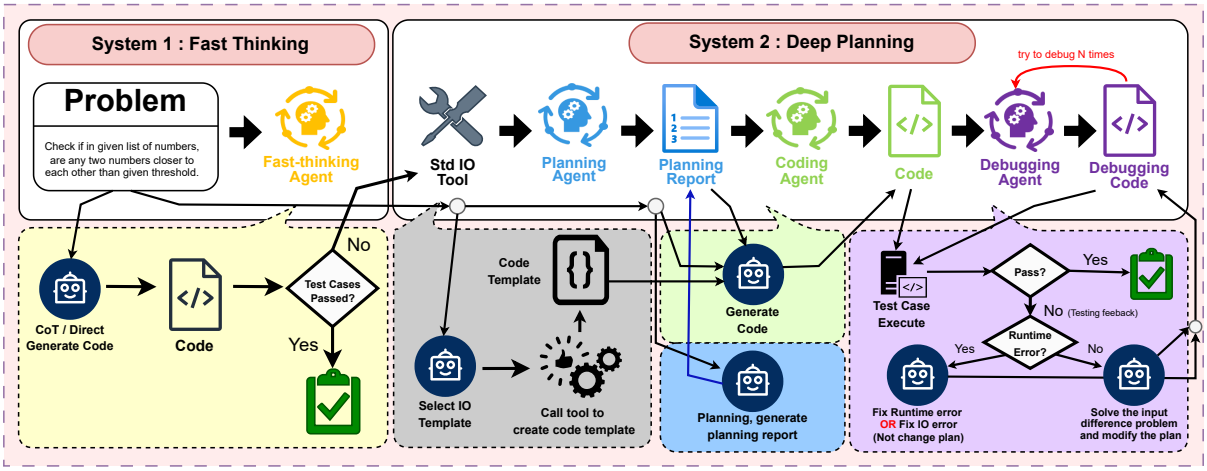

Figure 1: Overview of MACS-Coder

We evaluated MACS-Coder on the challenging LiveCodeBench V5 (Jain et al., 2024) and standard benchmarks (HumanEval, MBPP). Experiments using `gpt-oss-20B` (Agarwal et al., 2025) and Qwen3-series (Yang et al., 2025) demonstrate that MACS-Coder achieves new state-of-the-art (SOTA) results among open-source models. Crucially, it is competitive with the accuracy of heavy baselines like CodeSIM and MapCoder while dramatically improving computational efficiency, effectively narrowing the gap between mid-sized open models and elite proprietary systems.

Contributions. The main contributions of this work are summarized as follows:

- We analyze cost-performance trade-offs across single-agent, fixed multi-agent, and adaptive orchestration strategies on a spectrum of coding tasks.
- We introduce MACS-Coder, an adaptive orchestration framework that combines instance-wise difficulty estimation, progressive delegation, and early exit decisions.
- We demonstrate that structured I/O normalization is a first-order factor in competitive programming agents, rather than a peripheral engineering choice.
- We empirically demonstrate that MACS-Coder preserves high success rates on hard tasks while significantly reducing average computation on easy/medium tasks and that it improves the utility of small/medium open models when used as targeted specialists.
- We will release the MACS-Coder implementation and experimental configurations to support reproducibility and future research.

2 Related Work

LLMs for Code. The landscape of code generation has been transformed by both proprietary and open-source LLMs. Leading proprietary models include Anthropic's Claude series (Anthropic, 2024, 2025), OpenAI's GPT family (Achiam et al., 2023; OpenAI, 2025c,d), and Google's Gemini series (Comanici et al., 2025). Concurrently, the open-source ecosystem has flourished with powerful models such as Meta's Llama and Code Llama (Grattafiori et al., 2024; Meta AI, 2025; Rozière et al., 2023), Mistral's variants (Mistral AI, 2024b,a), DeepSeek's coding models (Shao et al., 2024; DeepSeek-AI et al., 2024; Guo et al., 2024), and Alibaba's Qwen series (Yang et al., 2024, 2025; Hui et al., 2024; Qwen Team, 2025). These models provide the foundational reasoning capabilities for modern code agents.

Multi-Agent Frameworks & Debugging. To address the limitations of single-turn generation, multi-agent frameworks have emerged. **MapCoder** (Islam et al., 2024) and **CodeSIM** (Islam et al., 2025) emulate human development cycles through planning, coding, and debugging stages. While effective, these systems often employ fixed, resource-intensive pipelines. Complementary research in debugging, such as **LDB** (Zhong et al., 2024) and **LPW** (Lei et al., 2025), focuses on runtime verification and plan-based refinement to isolate errors. MACS-Coder builds on these concepts but introduces an adaptive "Fast and Deep" architecture to optimize the trade-off between computational cost and problem-solving depth.

3 MACS-Coder

Our objective is to propose an efficient code agent through a two-stage "Fast and Deep Planning" design. The Fast-Thinking stage addresses simple problems that do not require extensive thought, while the Deep-Planning stage guides the LLM through step-by-step reasoning based on the processes of human engineers. By leveraging the concept of test-time compute, it allocates more resources to enhance the LLM's final performance. Inspired by software engineering practices, we designed a four-step LLM agent process for the Deep-Planning stage to simulate an engineer's workflow: analyzing the problem (Planning Agent), generating structured code (Coding Agent), debugging (Debugging Agent), and handling standard I/O (STD IO Tool). A detailed ablation study, presented in Section C, quantifies the significant impact of each of these components on both Pass@1 accuracy and token consumption. Drawing inspiration from recent works like MapCoder, CodeSIM, and LDB, we developed MACS-Coder to achieve a balance of performance and efficiency.

3.1 Fast Thinking System

The initial stage of our framework is the **Fast-Thinking System**, which serves as a rapid and resource-efficient filter. This stage is designed to handle problems for which the LLMs already exhibit high proficiency. The **Fast-Thinking Agent** generates a code solution with minimal guidance and evaluates its correctness against a set of unit tests.

This process acts as a simple yet effective **Test-time Heuristic** for difficulty assessment. We acknowledge that this is a pragmatic, test-time heuristic rather than an explicit learned difficulty model, functioning as an intrinsic **Difficulty Estimator**:

- If the generated code passes all unit tests, the solution is deemed correct and is returned immediately, minimizing computational expenditure.
- If any test fails, the problem is considered non-trivial and is escalated to the Deep-Planning System.

This dynamic routing mechanism ensures that MACS-Coder avoids the "Over-thinking" pitfall common in fixed-pipeline frameworks like CodeSIM, where simple tasks consume unnecessary resources.

Our implementation adapts its prompting strategy based on the problem domain; for complex competitive programming tasks (LiveCodeBench V5), the agent attempts direct code generation. For benchmarks with clearer solution patterns (HumanEval, MBPP), a lightweight planning-style prompt is used to improve initial accuracy.

3.2 Deep Planning System

This stage is designed for difficult problems or those where the LLM's responses are unstable. By increasing resource consumption, it enables the LLM to think more comprehensively and perform debugging to enhance its final performance.

3.2.1 Setup: STD-IO Tool

The STD-IO Tool is specifically designed for competitive programming problems, for which we created several input and output templates based on common formats. The LLM uses a tool-calling mechanism to select an appropriate template, ultimately producing a complete **code template**. This allows subsequent agents to modify the template directly when generating code rather than starting from scratch. Furthermore, our designed code templates improve the readability of the final code; details are provided in Appendix I. We show that structured I/O normalization is a first-order factor in competitive programming agents, rather than a peripheral engineering choice. Note that this tool is not activated for simpler benchmarks like HumanEval and MBPP, which lack complex I/O handling. The role of the STD-IO Tool within our overall workflow is illustrated in Figure 2 (highlighted as **Step 1**).

3.2.2 Strategy Formulation: Planning Agent

In the strategy formulation stage, the Planning Agent prompts the LLM to generate a **Planning Report** from the problem description. This report outlines the core algorithm, analyzes potential edge cases, and establishes a coding plan, providing a robust foundation for the implementation phase. To select an optimal algorithm, the LLM first enumerates all viable solutions and then chooses the most stable and straightforward strategy that prioritizes robustness and increases the likelihood of a correct solution. This stage corresponds to **Step 2** in our workflow, as depicted in Figure 2. (Full details of the instructional prompts are in Appendix I.2.)

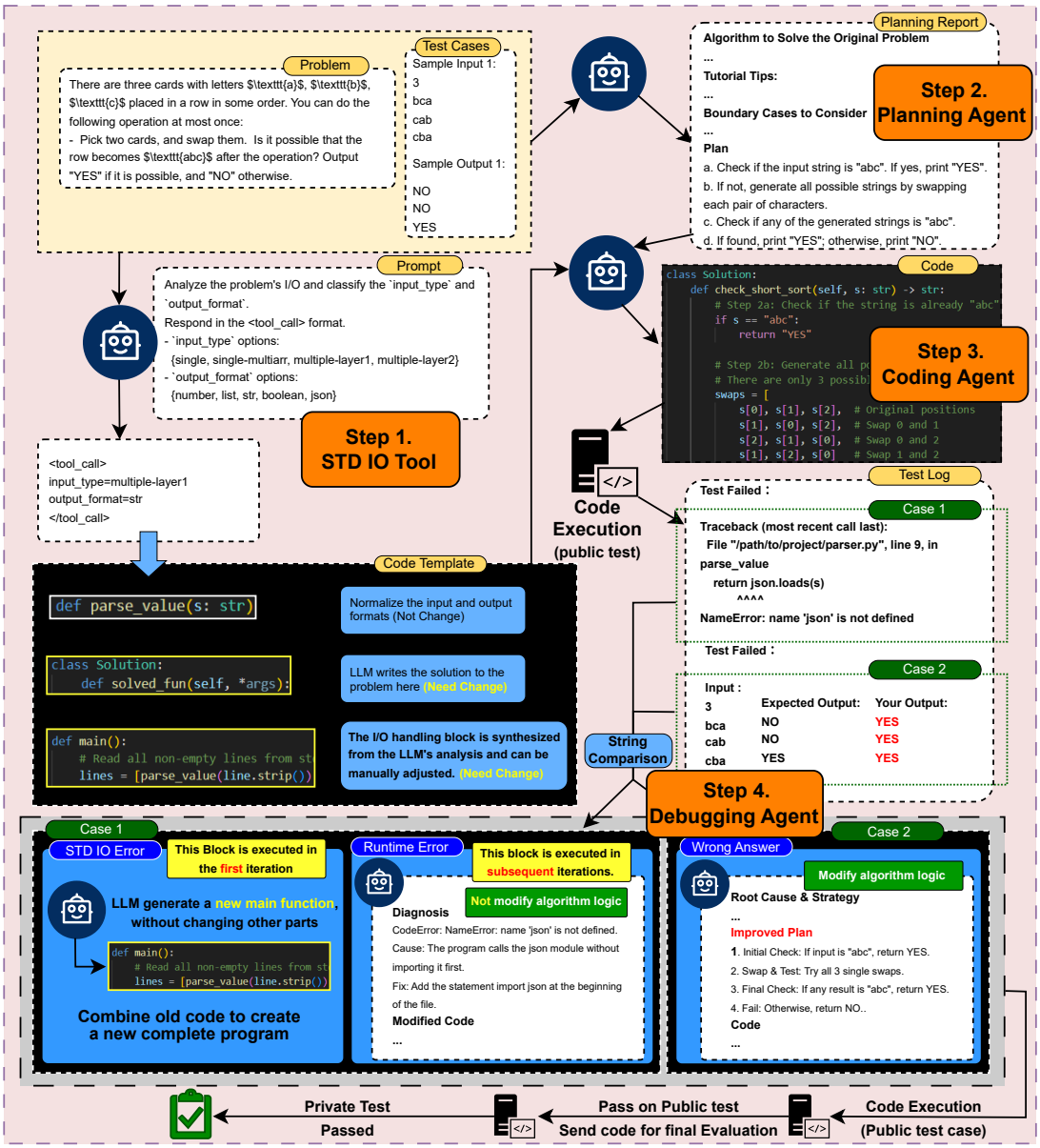

Figure 2: This figure illustrates the MACS-Coder pipeline in the Deep-Planning stage, featuring structured generation for stable, verifiable outputs and fine-grained debugging for precise error isolation, jointly improving efficiency and robustness in code generation.

3.2.3 Code Synthesis: Coding Agent

The Coding Agent (**Step 3**, Figure 2) synthesizes the final code. It takes the problem description, the Planning Report, and the Code Template as input, generating an implementation that conforms to the template's structure. To enforce adherence to the plan, the LLM is also prompted to generate code comments corresponding to each step in the Planning Report. The resulting code is then validated against a suite of unit tests. Code that passes all tests is considered successful; otherwise, it is passed to the Debugging Agent for refinement. (The prompts for this stage are detailed in Appendix I.3 and Appendix I.4.)

3.2.4 Refinement: Debugging Agent

In the Debugging Agent stage (**Step 4**, Figure 2), the agent receives the original problem, the Planning Report, the generated code, and the execution log from unit testing as input to debug the code. The Debugging Agent consists of three modules: the STD-IO Error Block, the Runtime Error Block, and the Wrong Answer Block. We first perform a string comparison on the execution log to determine if the code executed correctly. If it did, the process moves to the Output Error Block; otherwise, it is routed to either the Runtime Error Block

Contest-Level Problems (LiveCodeBench V5)											
LLM	**Approach**	**Easy**			**Medium**			**Hard**			**Total ACC (%)**
		ACC (%)	**Time (s)**	**Tokens**	**ACC (%)**	**Time (s)**	**Tokens**	**ACC (%)**	**Time (s)**	**Tokens**	
gpt-oss-20B	Direct	87.1	3.66	883	71.3	13.24	3,173	42.2	27.58	6,580	67.3
	CoT	87.8	3.07	721	68.3	10.48	2,465	38.9	23.59	5,459	65.4
	MapCoder	95.7	19.86	4,762	80.7	40.09	9,505	55.9	150.42	35,148	77.8
	CodeSIM	97.1	13.82	3,279	81.9	40.55	9,683	61.5	176.13	41,380	80.4
	MACS-Coder	**97.5**	**5.06**	**1,204**	**87.9**	**23.83**	**5,690**	**63.0**	**143.56**	**33,968**	**83.2**
Qwen3-14B	Direct	67.4	28.38	1,799	38.4	90.69	5,660	25.2	134.05	8313	43.5
	CoT	58.1	30.64	1,911	36.9	97.24	6,001	23.0	148.54	9,063	39.3
	MapCoder	87.8	82.82	5,144	65.0	**260.51**	**15,831**	32.6	**426.70**	**25,833**	62.2
	CodeSIM	91.8	142.54	8,622	61.6	467.18	28,107	28.5	726.95	44,037	61.0
	MACS-Coder	**95.3**	**64.69**	**3,971**	**72.8**	308.64	18,586	**35.9**	768.47	46,220	**68.6**
Qwen3-8B	Direct	61.3	32.46	2,072	39.0	51.57	3,231	20.7	60.68	3,748	40.4
	CoT	47.7	4.30	259	13.0	16.52	1,001	4.8	35.72	2,145	21.4
	MapCoder	72.0	**39.03**	**2,475**	31.1	**82.07**	**5,183**	10.7	**134.31**	**8,356**	52.0
	CodeSIM	87.1	89.34	5,688	53.2	262.34	16,592	22.2	678.91	41,591	54.4
	MACS-Coder	**94.6**	49.71	3,130	**66.5**	150.07	9,331	**26.7**	319.72	19,716	**63.1**

Table 1: Performance comparison of various methods on the LiveCodeBench V5 benchmark, utilizing state-of-the-art (SOTA) open-source SLMs including gpt-oss-20B, Qwen3-14B, and Qwen3-8B. We evaluate each approach based on three key metrics: ACC (Pass@1 Accuracy), Time (average seconds per problem), and Tokens (average generated response tokens per problem). Our proposed method, MACS-Coder, consistently achieves the highest accuracy across all tested models. **Red** indicates the best accuracy, and **Blue** indicates the best efficiency.

or the STD-IO Error Block. If this is the first execution failure, it enters the STD-IO Error Block; subsequent failures are handled by the Runtime Error Block.

STD-IO Error Block This block is entered on the first execution failure. Our experiments revealed that many execution failures are related to input parsing. If data is not read correctly, an error occurs. Therefore, this stage prompts the LLM to regenerate the code for reading and displaying input, while the core algorithmic code is preserved.

Runtime Error Block If the code fails to execute for a second time or more, this block is entered. It uses a more general prompt, asking the LLM to analyze the error itself and identify the cause of the failure to modify the code, again without altering the core problem-solving logic.

Wrong Answer Block If the code executes correctly but the output does not match the expected answer, this block is entered. Here, the LLM is prompted to analyze the incorrect output, simulate the failing test case, and generate an improved algorithm to modify the code.

(The specific debugging prompts for each block are presented in Appendix I.5.)

4 Experimental Setup

4.1 Evaluation Benchmarks and Baselines

We evaluate our method, MACS-Coder, on a diverse set of programming benchmarks. Our primary evaluation is on **LiveCodeBench V5** (Jain et al., 2024), a contamination-free benchmark of 880 competitive problems (279 Easy, 331 Medium, 270 Hard). For a comprehensive assessment of its performance on basic tasks, we present detailed results for **HumanEval** (Chen et al., 2021) (164 problems), **MBPP** (Austin et al., 2021) (974 problems), and their extended-test (-ET) versions (Dong et al., 2023) in Appendix D.1.

We compare MACS-Coder against several strong baselines: **Direct** generation (Chen et al., 2021), **Chain of Thought (CoT)** (Wei et al., 2022), the multi-agent **MapCoder** (Islam et al., 2024), and a state-of-the-art debugging framework, **CodeSIM** (Islam et al., 2025). We also include **LDB** (Zhong et al., 2024), although its use

is confined to the fundamental benchmarks (HumanEval/MBPP), as its framework is incompatible with the LiveCodeBench test format.

4.2 Implementation Details

Our core evaluation metric is **pass@1**. Our primary experiments utilize three models: **Qwen3 (8B, 14B)** and **gpt-oss (20B)**. To validate generalization, we present further experiments on a broader range of models, including **Llama3.1-8B**, **Gemma2-9B**, **Mistral-8B**, and **Qwen2.5-Coder-7B**, in Appendix D.2. Additionally, for our ablation studies on precision-efficiency trade-offs, we incorporated high-capacity models such as **GPT-5 mini** (OpenAI, 2025a) and **Gemini 2.0 Flash** (Google, 2024) to assess the framework's scalability and resource optimization.

Our MACS-Coder framework is governed by two key hyperparameters that control its nested-loop structure. These are p, the maximum number of outer *planning cycles*, and d, the maximum number of inner *debug attempts* per planning cycle. Key values for these hyperparameters and model-specific configurations used to suppress unwanted internal reasoning are detailed in Appendix B.

5 Results

In Table 1, we evaluate performance on complex competitive programming tasks. MACS-Coder demonstrates significant superiority, establishing a new SOTA with gpt-oss-20B achieving **83.2%** accuracy, clearly surpassing CodeSIM (80.4%) and MapCoder (77.8%).

Beyond accuracy, MACS-Coder excels in computational efficiency, achieving higher performance with fewer resources. For instance, on "Hard" problems with gpt-oss-20B, it reduces token consumption by nearly 18% compared to CodeSIM (33,968 vs. 41,380). This efficiency stems from our "Fast-Thinking" system, which strategically reserves deep reasoning resources for challenges that genuinely require them.

The framework's effectiveness is consistent across model scales. Notably, the Qwen3-8B model with MACS-Coder (63.1%) outperforms the larger Qwen3-14B using MapCoder (62.2%). These results confirm that MACS-Coder's dual-system architecture significantly enhances problem-solving capabilities across different model sizes.

5.1 Comparison with SOTA Models

To further validate the absolute competitiveness of the MACS-Coder framework, we compared it with the current strongest open-source and proprietary models on the LiveCodeBench benchmark. The data for this comparison is sourced from the official LiveCodeBench leaderboard (Jain et al., 2025), covering contest problems from **May 1, 2023, to February 1, 2025**. As shown in Table 2, the results indicate that MACS-Coder can elevate the capabilities of open-source small models to a level competitive with top-tier proprietary models.

The most striking result is that the gpt-oss-20B model equipped with MACS-Coder achieves a Pass@1 score of 83.2%. This performance surpasses several proprietary baselines reported on the same leaderboard, such as Grok-3-mini (81.4%) and o3-mini (80.5%), and remains within a close margin of the strongest models like Gemini-2.5 Pro (84.7%). This demonstrates that MACS-Coder is an extremely powerful performance enhancer, enabling a 20B-level open-source model to compete with the industry's top models.

Furthermore, the value of MACS-Coder lies in its ability to enable smaller models to achieve SOTA-level performance, making top-tier capabilities more accessible and deployable. The Qwen3-14B with MACS-Coder (68.6%) easily surpasses the un-enhanced gpt-oss-20B (67.3%) and is competitive with o1-mini (68.4%). Even the smaller Qwen3-8B, empowered by MACS-Coder (63.1%), outperforms a range of powerful models like DeepSeek V3 (56.3%) and Claude-3.5-Sonnet (51.5%). These results strongly demonstrate that the MACS-Coder framework provides an efficient pathway for the open-source community to tackle complex programming challenges that were previously only manageable by top-tier proprietary models, using relatively smaller models.

5.2 Performance on Live Codeforces Contests

We evaluated MACS-Coder on three live Codeforces Div. 2 contests to assess generalization. These contests feature original problems released after the model's knowledge cutoff, ensuring a strictly **leakage-free** evaluation unlike static benchmarks. For this experiment, we utilized the **gpt-oss-20B** model and modified the prompts to generate **C++** code instead of Python, optimizing for the strict time limits of competitive program-

LLM	LiveCodeBench V5			
	Easy	Medium	Hard	Pass@1
gpt-oss-120B (MACS-Coder)	97.9	90.6	**70.7**	**86.8**
o4-mini (High)	98.2	89.7	67.4	85.6
Gemini-2.5 Pro	98.2	**91.8**	61.9	84.7
gpt-oss-20B (MACS-Coder)	97.5	87.9	63.0	83.2
Grok-3-mini (High)	98.6	88.8	54.4	81.4
gpt-oss-120B	93.2	85.5	62.2	80.8
o3-mini (High)	**98.9**	88.5	51.5	80.5
o1 (Med)	**98.9**	84.6	49.3	78.3
DeepSeek-R1-Preview	**98.9**	84.8	47.7	77.9
Qwen3-14B (MACS-Coder)	95.3	72.8	35.9	68.6
o1-mini	95.0	73.4	34.8	68.4
gpt-oss-20B	87.1	71.3	42.2	67.3
Qwen3-8B (MACS-Coder)	94.6	66.5	26.7	63.1
DeepSeek V3	90.8	59.4	17.0	56.3
o1-Preview	94.3	56.8	14.1	55.6
Claude-3.5-Sonnet	92.9	46.3	14.9	51.5
Qwen3-14B	67.4	38.4	25.2	43.5
GPT-4o	87.4	36.8	6.1	43.4
Gemini-Pro-1.5	87.2	31.1	8.9	42.1
Qwen3-8B	61.3	39.0	20.7	40.4

Table 2: Comparison on LiveCodeBench V5. **MACS-Coder** significantly boosts Pass@1 scores. Best results are **bold**, second best underlined.

Contest	Participants	Rank	Percentile	Solved (Max Rating)
Round 1070	31,691	**202**	**0.64%**	A, B, C, D, **F (2750)**
Round 1069	17,078	864	5.06%	A, B, C, D
Round 1068	30,438	566	1.86%	A, B, C, D

Table 3: Performance of MACS-Coder on Live Codeforces Contests (Div. 2). Percentile is calculated based on total registered participants. The "Max Rating" indicates the highest difficulty level of the problem solved by the framework in that round.

ming.

As shown in Table 3, MACS-Coder demonstrated elite-level performance across all tested rounds. Most notably, in Round 1070, our framework achieved a rank of 202 out of 31,691 participants, placing it in the top 0.64% globally. Furthermore, the framework successfully solved Problem F, an “Expert-level” algorithmic challenge with a difficulty rating of 2750. In competitive programming, a 2750-rated problem typically represents the ’Master’ tier, solvable by less than 1% of global participants. These results underscore the robustness of MACS-Coder in handling high-stakes, competitive programming scenarios (see Appendix F for detailed verification).

6 Analyses

6.1 Performance-Cost Analysis

To visually demonstrate the efficiency advantage of MACS-Coder over the current SOTA method, CodeSIM, we plotted the relationship between accuracy and token consumption (as shown in Figure 3). The graph clearly indicates that MACS-Coder achieves both higher accuracy and lower token consumption across all benchmarks and models, establishing its significant superiority in performance efficiency.

The arrows in the figure consistently point from CodeSIM to MACS-Coder, always moving towards the top-left representing higher accuracy (y-axis) and lower token consumption (x-axis). In summary, while maintaining equivalent or superior accuracy, MACS-Coder saves on average **30%** of inference costs compared to fixed-pipeline baselines. This improvement is particularly pronounced in complex tasks. For example, on the LiveCodeBench benchmark, MACS-Coder not only increases the accuracy of Qwen3-8B by 8.7% but also saves over 10,000 tokens. Even with a more powerful model like gpt-oss-20B, MACS-Coder still improves accuracy by 2.8% while saving over 4,400 tokens. This comprehensive performance advantage validates the design of our "Fast and Deep Planning" framework. It intelligently allocates computational resources, avoiding unnecessary deep thought on simple problems to concentrate resources on tackling difficult ones. This makes MACS-Coder not only a more accurate solution but also a more economical and efficient framework for practical applications.

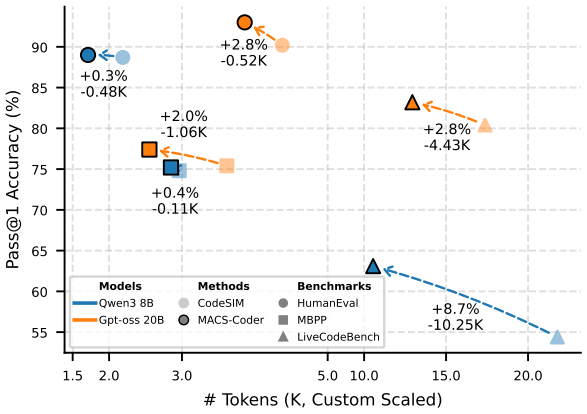

Figure 3: Accuracy-efficiency improvement achieved by our proposed methods. The plot shows that our approaches consistently push the models towards the ideal top-left corner (higher accuracy, fewer tokens).

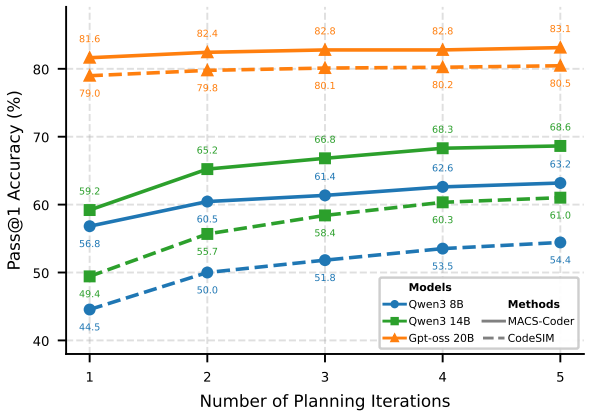

Figure 4: Performance comparison between **MACS-Coder** (solid lines) and CodeSIM (dashed lines) across multiple planning iterations. While both methods benefit from more iterations, MACS-Coder consistently operates at a higher performance level.

6.2 Analysis of Planning Iterations

To analyze the relationship between planning-stage investment and final performance, we evaluated the Pass@1 accuracy of MACS-Coder against CodeSIM over varying planning iterations (Figure 4). The results reveal a significant efficiency advantage for MACS-Coder, which consistently outperforms CodeSIM across all tested models and iteration counts. This efficiency gain is so substantial that MACS-Coder with the Qwen3-8B model at a single iteration achieves 56.8% Pass@1, surpassing the larger Qwen3-14B model using CodeSIM (49.4%). This finding demonstrates that our framework offers significant **"Capability Compression,"** enabling smaller models to achieve results previously requiring larger ones, effectively allowing them to punch above their weight class. This holds substantial implications for reducing computational costs while achieving state-of-the-art performance in practical applications.

6.3 Precision-Efficiency Trade-off Analysis

MACS-Coder demonstrates a superior precision-efficiency balance on LiveCodeBench v5, as shown in Figure 5. While occasionally solving marginally fewer problems (1-3 instances) than CodeSIM on high-capacity models, it reduces token consumption by 15–30%. Notably, on Gemini 2.0 Flash, it surpasses CodeSIM (46.0% vs. 34.5%) using only one-third of the tokens. This demonstrates high **Intelligence Leverage**: the framework amplifies the potential of strong models rather than merely patching weaknesses. This underscores MACS-Coder's practical advantage in minimizing costs while maintaining SOTA performance (see Appendix E).

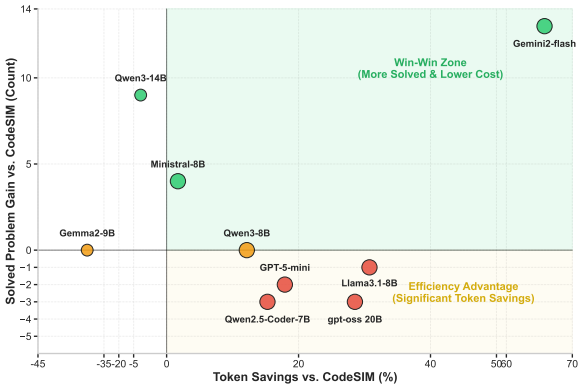

Figure 5: MACS-Coder Advantage Analysis. This figure highlights the superior balance between algorithmic precision and computational efficiency achieved by MACS-Coder compared to CodeSIM across various models.

7 Conclusion and Future Work

In this paper, we introduced MACS-Coder, a novel framework designed to solve complex programming tasks. At its core, MACS-Coder features an innovative "Fast and Deep Planning" dual-system architecture that adaptively adjusts its strategy based on problem difficulty and integrates structured code template generation with a fine-grained debugging agent. Evaluation results across multiple mainstream code generation benchmarks show that MACS-Coder significantly surpasses existing SOTA methods in both accuracy and computational efficiency. Our research demonstrates that by empowering small open-source models, MACS-Coder successfully elevates their performance to a level competitive with top-tier proprietary models.

Future work will focus on extending this framework to other domains requiring complex reasoning, such as mathematical problem-solving and general question-answering, to broaden its scope and impact.

Limitations

Despite the strong performance and efficiency of MACS-Coder, some limitations exist. First, while our framework demonstrates strong generalization across multiple open-source SLMs, the magnitude

of performance improvement is not entirely consistent. This variability is partly tied to the intrinsic capabilities of the base models themselves; an agentic framework can structure and guide an SLM's reasoning, but its effectiveness is ultimately constrained by the model's fundamental coding ability. If a model's baseline performance is below a certain threshold, the framework's capacity to elicit further gains is limited. Additionally, adapting our structured prompts for specific model architectures remains an area for optimization. Second, while MACS-Coder is more token-efficient than prior SOTA methods, its computational cost remains higher than Direct Prompting, marking an avenue for future work. Finally, the efficacy of our debugging agent is highly dependent on the quality and coverage of the provided test cases.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, and 1 others. 2025. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2501.00000*.

Anthropic. 2024. The claude 3 model family: Opus, sonnet, haiku. *arXiv preprint arXiv:2403.05530*.

Anthropic. 2025. Introducing claude 4. https://www.anthropic.com/news/claude-4. Accessed: 2025-09-01.

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

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé, Jared Kaplan, Harrison Edwards, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, and 1 others. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bing-Li Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, and 1 others. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Yihong Dong, Ji Ding, Xue Jiang, Zhuo Li, Ge Li, and Zhi Jin. 2023. Codescore: Evaluating code generation by learning code execution. *ACM Transactions on Software Engineering and Methodology*, 34(1):1–22.

Google. 2024. Gemini 2.0 Flash model documentation. https://docs.cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/2-0-flash. Accessed: 2026-01-02.

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

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming - the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, 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, Bangkok, Thailand. Association for Computational Linguistics.

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, Albuquerque, New Mexico. Association for Computational Linguistics.

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*.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. Livecodebench leaderboard. https://livecodebench.github.io/leaderboard_v5.html. Accessed: 2025-09-01.

Chao Lei, Yanchuan Chang, Nir Lipovetzky, and Krista A. Ehinger. 2025. Planning-driven programming: A large language model programming workflow. In *Proceedings of the 63rd Annual Meeting of*

the Association for Computational Linguistics (Volume 1: Long Papers), pages 12647–12684, Vienna, Austria. Association for Computational Linguistics.

Wenjie Ma, Jingxuan He, Charlie Snell, Tyler Griggs, Sewon Min, and Matei Zaharia. 2025. Reasoning models can be effective without thinking. *arXiv preprint arXiv:2504.09858*.

Meta AI. 2025. The llama 4 herd: The beginning of a new era of natively multimodal ai innovation. https://ai.meta.com/blog/llama-4-multimodal-intelligence/. Accessed: 2025-09-01.

Mistral AI. 2024a. Codestral. https://mistral.ai/news/codestral. Accessed: 2025-09-07.

Mistral AI. 2024b. Mistral large. https://mistral.ai/news/mistral-large. Accessed: 2025-09-01.

OpenAI. 2025a. GPT-5-mini model documentation. https://platform.openai.com/docs/models/gpt-5-mini. Accessed: 2026-01-02.

OpenAI. 2025b. GPT-OSS 20B hugging face model card. https://huggingface.co/openai/gpt-oss-20b. Accessed: 2025-09-01.

OpenAI. 2025c. Introducing GPT-5. https://openai.com/gpt-5/. Accessed: 2025-09-01.

OpenAI. 2025d. Introducing OpenAI o3 and o4-mini. https://openai.com/index/introducing-o3-and-o4-mini/. Accessed: 2025-09-07.

Qwen Team. 2025. Qwen3-coder: Agentic coding in the world. https://qwenlm.github.io/blog/qwen3-coder/. Accessed: 2025-09-07.

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

Zhihong Shao, Damai Dai, Daya Guo, Bo Liu, Zihan Wang, and Huajian Xin. 2024. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837.

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

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, and 1 others. 2024. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*.

Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Debug like a human: A large language model debugger via verifying runtime execution step by step. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3274–3293, Bangkok, Thailand. Association for Computational Linguistics.

A Algorithm of MACS-Coder

Algorithm 1 outlines the complete workflow of the MACS-Coder framework. The process begins with the **Fast Thinking** stage, where the model attempts to generate a solution directly. If this initial solution passes the provided unit tests, it is returned immediately to minimize computational costs. If the fast solution fails, the framework escalates to the **Deep Planning** stage. This stage operates as a nested loop structure: an outer loop controlled by parameter p governs the generation of new strategic plans and code templates, while an inner loop controlled by parameter d manages the iterative debugging process. Within the debugging loop, the system analyzes execution logs to classify errors (Runtime or Wrong Answer) and applies targeted fixes. This dual-stage, iterative design ensures that simple problems are solved efficiently while complex problems receive the necessary depth of reasoning and refinement.

Algorithm 1 MACS-Coder

Input: Problem P, Unit Tests U
Output: Solution C

▷ **Stage 1: Fast Thinking**
$C \leftarrow \text{FastGen}(P)$
if $\text{Test}(C, U)$ **then return** C
end if

▷ **Stage 2: Deep Planning**
for $i \leftarrow 1$ to p **do** ▷ Planning cycles (p)
  $T \leftarrow \text{GenTemplate}(P)$; $R \leftarrow \text{GenPlan}(P)$
  $C \leftarrow \text{GenCode}(P, R, T)$

  for $j \leftarrow 1$ to d **do** ▷ Debug attempts (d)
    $passed, log \leftarrow \text{Test}(C, U)$
    if $passed$ **then return** C
    end if
    $type \leftarrow \text{Analyze}(log)$
    if $type$ is Runtime **then**
      $C \leftarrow \text{FixRuntime}(P, C, log)$
    else if $type$ is WrongAnswer **then**
      $C \leftarrow \text{FixLogic}(P, R, C, log)$
    end if
  end for
end for
return C

B Hyperparameter Configuration

In this section, we provide the detailed hyperparameter settings used in our experiments. The MACS-Coder framework utilizes a nested-loop structure controlled by two primary parameters: p (planning cycles) and d (debug attempts). Table 4 outlines the specific values used for different model categories. For primary models like Qwen3 and gpt-oss, we set $p = 5$ and $d = 5$ to maximize reasoning depth. For generalization models, we used a lighter configuration with $p = 1$ and $d = 5$. Additionally, specific prompt engineering strategies were employed to suppress internal chain-of-thought behaviors in certain models to ensure adherence to our structured planning process.

C Ablation Study

C.1 Effectiveness of the Dual-System

To validate the value of our "Fast and Deep Planning" dual-system architecture, we conducted a detailed comparison across different models, with results shown in Table 5. The findings highlight two critical insights regarding the Fast Thinking System.

First, even without the Fast Thinking module, MACS-Coder consistently outperforms the baseline CodeSIM in both accuracy and token efficiency. As shown in the table, for `gpt-oss-20b`, MACS-Coder (w/o FastThinking) achieves an accuracy of 83.0%, surpassing CodeSIM's 80.45%, while also consuming fewer tokens (14,043 vs. 17,377). A similar trend is observed with `Qwen3-8B` and `Qwen3-14B`, where the Deep Planning System alone proves superior to the fixed pipeline of CodeSIM. This confirms that our core planning and debugging architecture is robust and effective on its own.

Second, integrating the Fast Thinking System further optimizes the trade-off between performance and computational cost. For `Qwen3-8B`, enabling Fast Thinking dramatically boosts Pass@1 accuracy from 55.23% to 63.18% while reducing average token usage from 11,505 to 10,551. This demonstrates the system's ability to efficiently solve simpler problems without over-reasoning. While `Qwen3-14B` shows a slight dip in accuracy with Fast Thinking (68.64% vs. 69.20%), it benefits from a reduction in token costs. Overall, the dual-system design ensures that resources are allocated intelligently, maximizing efficiency without compromisingand often enhancingperformance.

C.2 Agent Contributions

To quantify the contribution of the core agents in the "Deep-Planning" system, we conducted further ablation studies, with results shown in Table 6. The data clearly reveals the critical impact of the STD IO Tool and the Debugging Agent on final performance. When we remove the Debugging

Model Category	Parameters (p, d)	Configuration Detail
Primary Models	$p = 5, d = 5$	Qwen3: "Non-Thinking Mode" + specific prompt (Ma et al., 2025). gpt-oss: “Reasoning:Low” prompt (OpenAI, 2025b).
Generalization Models	$p = 1, d = 5$	N/A

Table 4: Hyperparameter and model-specific configurations.

Model	Method	Pass@1 (%)	Avg Tokens
gpt-oss-20b	CodeSIM	80.45	17,377
	MACS-Coder (w/o FT)	83.00	14,043
	MACS-Coder	**83.30**	**12,943**
Qwen3-14B	CodeSIM	61.02	26,816
	MACS-Coder (w/o FT)	**69.20**	24,225
	MACS-Coder	68.64	**22,430**
Qwen3-8B	CodeSIM	54.43	20,805
	MACS-Coder (w/o FT)	55.23	11,505
	MACS-Coder	**63.18**	**10,551**

Table 5: Performance comparison of MACS-Coder with and without Fast Thinking against CodeSIM on LiveCodeBench V5.

Agent, the Pass@1 accuracy drops from 63.1% to 55.3%, a performance loss of 7.8%. Removing the STD IO Tool leads to an even greater performance decline, with accuracy falling to 53.6%, a drop of 9.5%. If both are disabled, the accuracy plummets to 47.8%. This means that each agent contributes approximately 8-9% to the accuracy, which, in a large test set, is equivalent to successfully solving 70 to 80 additional problems. This result strongly demonstrates that our structured template generation and fine-grained debugging processes, designed to mimic human engineers, are the two pillars that enable MACS-Coder to stably solve complex problems.

STD IO Tool	Debugging Agent	Pass@1	Performance Drop
✗	✗	47.8%	15.3%
✓	✗	55.3%	7.8%
✗	✓	53.6%	9.5%
✓	✓	63.1%	-

Table 6: Ablation study on STD IO Tool and Debugging Agent (on LiveCodeBench V5). ✓ indicates the component is enabled, and ✗ indicates it is disabled.

C.3 Debugging Modules Analysis

To validate the value of each specialized module within our fine-grained Debugging Agent, we conducted an ablation study, with results shown in Table 7. The data indicates that every error-handling module makes an indispensable contribution to the final performance. The module for handling **Runtime Errors** has the most significant impact; its removal leads to a 5.0% performance drop, highlighting the importance of managing runtime exceptions in complex programming. In comparison, the modules for handling **STD IO Errors** and **Wrong Answer** have a smaller but still crucial impact (0.5% and 0.9%, respectively), and their presence collectively ensures the integrity and robustness of the debugging process. This result proves that our fine-grained design, which allows the agent to apply the most appropriate correction strategy for different failure reasons, is key to its efficient debugging capabilities.

Runtime Error	STD IO Error	Wrong Answer	Pass@1	Performance Drop (%)
✓	✓	✓	63.1%	-
✗	✓	✓	58.1%	5.0%
✓	✗	✓	62.6%	0.5%
✓	✓	✗	62.2%	0.9%

Table 7: Ablation study of Debugging Agent components (on LiveCodeBench V5). ✓ indicates the component is enabled, and ✗ indicates it is disabled.

D Additional Experiments

D.1 Basic Code Generation

In the basic code generation tasks shown in Table 8, the core advantage of MACS-Coder is its outstanding overall performance efficiency, achieving an optimal balance between top-tier accuracy and computational cost.

The efficient design of MACS-Coder allows it to reach SOTA-level accuracy with fewer resources. For example, on HumanEval, the Qwen3-

Basic Programming Problems											
LLM	**Approach**	**HumanEval**					**MBPP**				
		ACC (%)	**Time (s)**	**Tokens**	**ET ACC (%)**	**Avg ACC (%)**	**ACC (%)**	**Time (s)**	**Tokens**	**ET ACC (%)**	**Avg ACC (%)**
gpt-oss 20B	Direct	62.2	4.07	981	55.5	58.8	47.6	3.51	848	31.2	39.4
	CoT	98.2	3.49	829	85.4	91.8	77.6	2.15	522	52.6	65.1
	LDB	98.8	**3.91**	**937**	86.0	92.4	90.7	**4.53**	**1096**	58.7	74.7
	MapCoder	97.0	22.31	5314	84.8	90.9	92.9	21.04	5047	61.2	77.0
	CodeSIM	97.0	18.43	4377	83.5	90.2	91.4	15.03	3616	59.4	75.4
	MACS-Coder	**99.4**	16.23	3861	**86.6**	**93.0**	**93.2**	10.69	2553	**61.7**	**77.4**
Qwen3-14B	Direct	48.2	14.16	910	43.3	45.7	40.1	14.16	911	30.0	35.0
	CoT	93.9	15.86	1000	**84.1**	89.0	76.6	14.25	912	51.6	64.1
	LDB	95.1	**23.21**	**1465**	**84.1**	89.6	90.9	**36.09**	**2263**	**62.5**	**76.7**
	MapCoder	94.5	46.16	2896	81.7	88.1	88.9	79.38	4851	59.2	74.0
	CodeSIM	95.1	89.62	5460	**84.1**	89.6	**92.4**	88.72	5413	59.7	76.0
	MACS-Coder	**95.7**	44.87	2812	**84.1**	**89.9**	90.7	67.33	4181	59.9	75.3
Qwen3-8B	Direct	48.8	25.23	1621	41.5	45.1	20.4	23.21	1500	14.6	17.5
	CoT	79.3	1.54	96	70.1	74.7	74.1	1.42	92	51.9	63.0
	LDB	88.4	**11.80**	**758**	78.0	83.2	84.1	**12.07**	**775**	55.2	69.6
	MapCoder	91.5	26.29	1685	78.0	84.7	86.1	29.57	1746	56.2	71.1
	CodeSIM	**94.5**	35.09	2188	82.9	88.7	89.2	47.68	2962	**60.5**	74.8
	MACS-Coder	93.9	26.80	1711	**84.1**	**89.0**	**89.9**	48.24	2849	**60.5**	**75.2**

Table 8: Performance comparison on the HumanEval and MBPP benchmarks using open-source models (gpt-oss-20B, Qwen3-14B, and Qwen3-8B). The evaluation metrics include: ACC (Pass@1 Accuracy), Time (average seconds per problem), Tokens (average generated response tokens per problem), and ET ACC (a more stringent evaluation with additional private test cases). The results show that MACS-Coder not only achieves the highest accuracy but also demonstrates superior efficiency by consuming fewer Tokens than other high-performing methods like CodeSIM and MapCoder. **Red** highlights the best accuracy, while **Blue** highlights the best efficiency.

8B model with MACS-Coder not only achieves a top average accuracy of 89.0% but also has significantly lower time (26.80s) and token (1,711) consumption compared to the similarly performing CodeSIM.

More importantly, this high efficiency enables smaller models to achieve performance beyond their scale. The Qwen3-8B with MACS-Coder on HumanEval (89.0%) performs on par with the larger Qwen3-14B using the CoT method (89.0%). This proves that our framework can achieve results with more economical smaller models that previously required more expensive larger models, offering a solution for the code generation field that combines top-tier capability with practical utility.

D.2 Performance Across Open-source SLMs

To further demonstrate the generalization capability of the MACS-Coder framework, we evaluated its performance across several mainstream open-source SLMs, including Llama3.1-8B, Gemma2-9B, Ministral-8B, and Qwen2.5-Coder-7B. As shown in Table 9, the results highlight MACS-Coder's exceptional adaptability and its superior balance between accuracy and efficiency.

On the more complex **LiveCodeBench V5** dataset, MACS-Coder consistently and significantly outperforms CodeSIM in accuracy across all tested models. For instance, with **Ministral-8B**, MACS-Coder achieves an accuracy of 20.0%, a substantial improvement over CodeSIM's 14.0%. Similar advantages are observed on **Llama3.1-8B** (15.1% vs. 13.9%) and **Gemma2-9B** (12.5% vs. 10.3%), proving its robust problem-solving capabilities.

On the **HumanEval** dataset, MACS-Coder demonstrates a more nuanced but equally compelling advantage. While its accuracy is highly competitive—and even superior on models like **Ministral-8B** (85.4% vs. 79.3%)—its primary strength lies in its remarkable efficiency. For example, on **Qwen2.5-Coder-7B**, although CodeSIM's accuracy is slightly higher (87.8% vs. 86.6%), MACS-Coder completes the task **29% faster** while consuming **25% fewer tokens**. This

trend of achieving comparable or higher accuracy with significantly less computational cost is a consistent theme across the models.

E Performance Analysis on Recent LiveCodeBench Problems

To rigorously validate the robustness and generalization capability of MACS-Coder, we conducted an additional evaluation using a subset of LiveCodeBench problems released **after November 1, 2024**. This timeframe postdates the training cutoff of all evaluated models, ensuring a strictly contamination-free environment to test true out-of-distribution generalization. The dataset comprises 113 problems (31 Easy, 36 Medium, 46 Hard), representing the latest trends in competitive programming.

E.1 Results and Analysis

Table 10 details the performance comparison across varying difficulty levels. The results provide critical insights into the architectural advantages of MACS-Coder over fixed-pipeline approaches like CodeSIM.

MACS-Coder exhibits its strongest advantage in medium-difficulty problems, a domain where strategic planning yields the highest return on investment. On **Qwen-14B**, our framework achieves **52.8%** accuracy compared to CodeSIM's 38.9%, a substantial **+13.9% improvement**. Similarly, on **Qwen-8B**, MACS-Coder leads by 2.8 percentage points. This pattern confirms that the dual-system architecture effectively identifies problems requiring moderate reasoning depthallocating necessary resources without the overhead of full-scale brute force.

For easy-difficulty problems, MACS-Coder matches or exceeds the SOTA performance of CodeSIM while maintaining the low computational footprint detailed in our main analysis. Notably, on **Qwen-14B**, MACS-Coder achieves a near-perfect **96.8%** accuracy (30/31 solved) versus CodeSIM's 83.9%. This validates the "Fast-Thinking" module's ability to swiftly handle straightforward tasks, preventing the over-engineering often seen in static multi-agent frameworks.

On hard-difficulty tasks, the results highlight a nuanced trade-off between absolute accuracy and operational efficiency. While CodeSIM holds a slight edge on **GPT-oss-20B** (63.0% vs. 58.7%), this marginal gain comes at a disproportionate computational cost. As noted in our main analysis, MACS-Coder reduces token consumption by over 18% in this tier. The slight performance gap suggests that CodeSIM's gains rely heavily on aggressive, resource-intensive trial-and-error. In contrast, MACS-Coder prioritizes a more stable, verifiable reasoning path, offering a more practical solution for real-world deployment where resource constraints matter.

A standout finding is the synergy with **Qwen-14B**, where MACS-Coder achieves **55.8%** overall accuracy, significantly outperforming CodeSIM (47.8%) by **8.0 percentage points**. This indicates that our structured reasoning framework is particularly effective at "unlocking" the latent planning capabilities of mid-sized models that are often underutilized by direct prompting or less adaptive agents. The framework proves its scalability even on smaller architectures. On **Ministral-8B**, MACS-Coder (15.9%) outperforms both CodeSIM (12.4%) and Direct prompting (9.7%). While absolute scores on these smaller models are lower, the consistent relative improvement confirms that MACS-Coder's benefits are architectural and not limited to specific model families.

The evaluation on unseen, recent problems strongly supports the core thesis of MACS-Coder: adaptive, difficulty-aware orchestration yields superior performance-efficiency trade-offs. The framework excels in the "sweet spot" of medium-difficulty tasks and maintains high efficiency on easier ones. While extremely hard problems present a challenge, MACS-Coder offers a balanced alternative to the prohibitive costs of exhaustive search methods, making it a highly viable candidate for scalable, production-grade code generation.

F Live Codeforces Contest Verification

This section provides the raw verification data for the live Codeforces experiments discussed in the main text. Figure 6 presents the official ranking screenshots for Rounds 1068, 1069, and 1070. These images serve to authenticate the performance metrics reported in Table 3, confirming both the final rankings and the specific problems solved. The submission logs further verify that all solutions were generated within the strict real-time constraints of the live contests, ensuring a

LLM	Approach	HumanEval				LiveCodeBench V5		
		ACC (%)	Time (s)	Tokens	ET ACC (%)	ACC (%)	Time (s)	Tokens
Llama3.1 8B Instruct	Direct	50.0	5.58	395	43.3	10.7	6.51	451
	CodeSIM	**82.3**	69.12	4650	**71.3**	13.9	94.94	6304
	MACS-Coder	80.5	**56.92**	**3881**	67.7	**15.1**	**63.69**	**4271**
Gemma2 9B Instruct	Direct	66.5	8.98	477	58.5	12.0	10.54	537
	CodeSIM	80.5	87.35	**3981**	67.1	10.3	**84.09**	**3870**
	MACS-Coder	**81.1**	**83.00**	4029	**67.7**	**12.5**	110.01	5165
Ministral 8B Instruct	Direct	74.4	6.41	447	64.6	13.9	8.64	579
	CodeSIM	79.3	33.29	2161	67.7	14.0	85.40	5214
	MACS-Coder	**85.4**	**27.00**	**1760**	**71.3**	**20.0**	**77.78**	**4803**
Qwen2.5-Coder 7B Instruct	Direct	70.1	3.31	233	62.8	18.5	6.08	426
	CodeSIM	**87.8**	25.07	1661	**76.8**	23.0	74.85	5207
	MACS-Coder	86.6	**17.70**	**1246**	75.0	**23.3**	**64.09**	**4427**

Table 9: Performance comparison on HumanEval and LiveCodeBench V5 across different open-source LLMs. **Red** indicates the best accuracy, and **Blue** indicates the best efficiency.

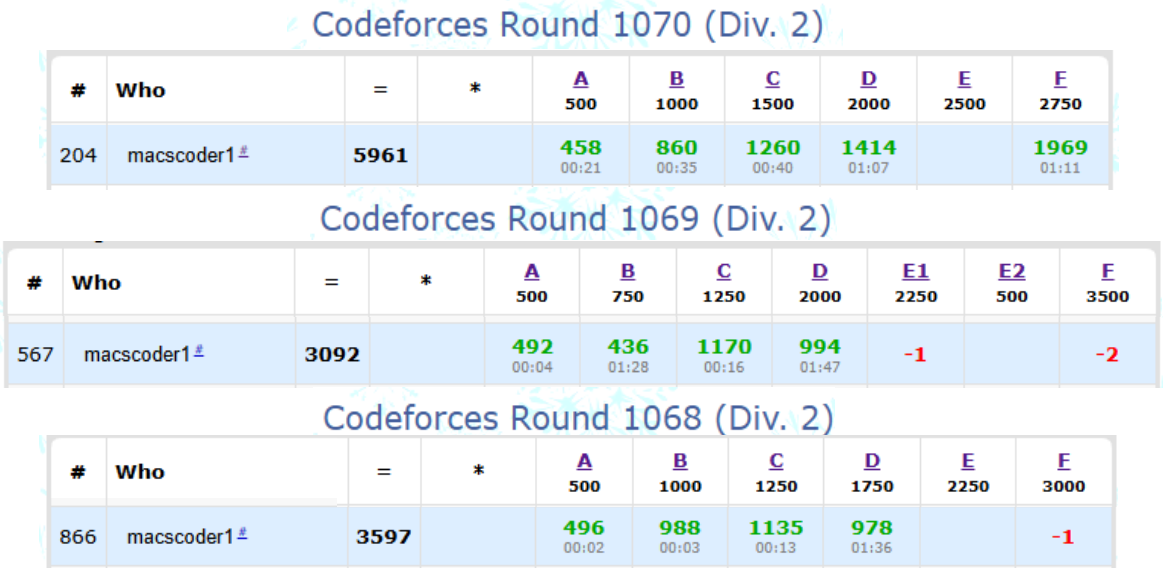

Figure 6: Verification snapshots from the Codeforces platform. This image serves as empirical evidence for the data reported in Table 3, illustrating MACS-Coder's ability to handle novel, complex algorithmic problems in a zero-shot, real-time setting.

contamination-free evaluation environment.

G Case Study

G.1 Robustness in I/O Handling

While direct generation (without agentic planning) is capable of producing correct core algorithms, particularly for **Easy** problems in LiveCodeBench, it often fails in competitive environments due to fragile I/O handling. We compare the Direct output and MACS-Coder output for the task *Lexicographically Smallest Palindrome* (Task 2816), as shown in Figure 7.

G.2 Semantic Reasoning in Hard Problems

For **Hard** problems in LiveCodeBench, which often require complex logical deductions beyond simple pattern matching, direct generation frequently misinterprets the core mathematical constraints. We illustrate this with Task 3047 (*Find the Maximum Sum of a Complete Set*), a hard-difficulty problem requiring number theory insights, as shown in Figure 8.

Contest-Level Problems (LiveCodeBench V5 - Recent)						
LLM	**Approach**	**Easy (31)**	**Medium (36)**	**Hard (46)**	**Total (113)**	**Avg Tokens**
		ACC (Solved)	**ACC (Solved)**	**ACC (Solved)**	**ACC (Solved)**	
GPT-5 mini	Direct	93.6 (29)	**88.9 (32)**	58.7 (27)	77.9 (88)	9,302
	CodeSIM	**96.8 (30)**	83.3 (30)	**78.3 (36)**	**85.0 (96)**	32,566
	MACS-Coder	93.6 (29)	**88.9 (32)**	71.7 (33)	83.2 (94)	**26,729**
Gemini 2.0 Flash	Direct	61.3 (19)	19.4 (7)	23.9 (11)	32.7 (37)	405
	CodeSIM	58.1 (18)	27.8 (10)	23.9 (11)	34.5 (39)	53,448
	MACS-Coder	**83.9 (26)**	**36.1 (13)**	**28.3 (13)**	**46.0 (52)**	**18,287**
gpt-oss 20B	Direct	90.3 (28)	69.4 (25)	34.8 (16)	61.1 (69)	5,008
	CodeSIM	**96.8 (30)**	**77.8 (28)**	**63.0 (29)**	**77.0 (87)**	23,593
	MACS-Coder	**96.8 (30)**	75.0 (27)	58.7 (27)	74.3 (84)	**16,859**
Qwen3-14B	Direct	67.7 (21)	19.4 (7)	23.9 (11)	34.5 (39)	6,623
	CodeSIM	83.9 (26)	38.9 (14)	30.4 (14)	47.8 (54)	**30,526**
	MACS-Coder	**96.8 (30)**	**52.8 (19)**	**30.4 (14)**	**55.8 (63)**	31,727
Qwen3-8B	Direct	58.1 (18)	27.8 (10)	15.2 (7)	31.0 (35)	3,243
	CodeSIM	83.9 (26)	33.3 (12)	**23.9 (11)**	**43.4 (49)**	18,055
	MACS-Coder	**87.1 (27)**	**36.1 (13)**	19.6 (9)	**43.4 (49)**	**15,860**
Gemma2-9B	Direct	**25.8 (8)**	2.8 (1)	4.3 (2)	9.7 (11)	568
	CodeSIM	22.6 (7)	**5.6 (2)**	4.3 (2)	9.7 (11)	**4,137**
	MACS-Coder	**25.8 (8)**	2.8 (1)	4.3 (2)	9.7 (11)	5,690
Llama3.1-8B	Direct	22.6 (7)	5.6 (2)	0.0 (0)	8.0 (9)	470
	CodeSIM	**29.0 (9)**	**5.6 (2)**	**2.2 (1)**	**10.6 (12)**	6,693
	MACS-Coder	**29.0 (9)**	**5.6 (2)**	0.0 (0)	9.7 (11)	**4,635**
Ministral-8B	Direct	25.8 (8)	2.8 (1)	4.3 (2)	9.7 (11)	617
	CodeSIM	29.0 (9)	**8.3 (3)**	4.3 (2)	12.4 (14)	5,540
	MACS-Coder	**38.7 (12)**	**8.3 (3)**	**6.5 (3)**	**15.9 (18)**	**5,445**
Qwen2.5-Coder-7B	Direct	32.3 (10)	**8.3 (3)**	4.3 (2)	13.3 (15)	485
	CodeSIM	**48.4 (15)**	5.6 (2)	**6.5 (3)**	**17.7 (20)**	5,868
	MACS-Coder	35.5 (11)	**8.3 (3)**	**6.5 (3)**	15.0 (17)	**4,971**

Table 10: Detailed performance comparison on recent LiveCodeBench problems (released after 2024-11-01). The table reports Pass@1 accuracy (%) for Easy, Medium, Hard, and Total categories, along with the average number of tokens generated. Our proposed method, MACS-Coder, consistently achieves competitive accuracy with superior efficiency. **Red** marks the highest accuracy, and **Blue** marks the best efficiency.

H Role of LLMs

We utilized LLMs (e.g., GPT-5, Gemini 2.5) solely for grammatical polishing and generating boilerplate code. All scientific concepts, experimental designs, and analyses are the original work of the authors.

Direct Generation (Failed)

```
# Fragile Input Parsing
if __name__ == "__main__":
    import sys
    s = sys.stdin.readline().strip()
    # Manual and limited quote stripping
    if len(s) >= 2 and ((s[0] == s[-1] == '"')):
        s = s[1:-1]
    sol = Solution()
    # Missing required output quotes
    print(sol.makeSmallestPalindrome(s))
```

MACS-Coder (Success)

```
# Robust Input Parsing via parse_value
def main():
    # Systematic parsing using json.loads
    lines = [parse_value(line.strip())
             for line in sys.stdin if line.strip()]
    args = [parse_value(line) for line in lines]
    result = Solution().solve(*args)
    # Correctly matched output format
    print(f'"{result}"')
```

Comparison Analysis

The Direct method correctly implemented the greedy algorithm but failed to match the expected output format (quoted string). Direct generation fails the 'last-mile' integration due to hard-coded I/O parsing, whereas MACS-Coders STD-IO Tool leverages schema-aware templates to ensure seamless environment adaptation. In contrast, MACS-Coder's `stdio_check` module identified the JSON-formatted string requirement, utilizing a systematic `parse_value` template and correctly formatted print statement.

Figure 7: Comparison of Driver Code between Direct Generation and MACS-Coder. Despite identical core logic, MACS-Coder succeeds by accurately adapting to the competition's I/O specifications.

Direct Model (Logical Failure)

```
# WRONG: Factorizing array VALUES
sum_of_part = dict()
for v in nums:
    x = v
    for p in primes:
        if p * p > x: break
        while x % (p * p) == 0:
            x //= p * p
    key = x # key derived from nums[i]
    sum_of_part[key] = sum_of_part.get(key, 0) + v
```

MACS-Coder (Logical Success)

```
# CORRECT: Factorizing array INDICES
sums_by_key = defaultdict(int)
for idx, val in enumerate(nums, start=1):
    # key derived from index 'idx'
    key = key_of_idx(idx)
    sums_by_key[key] += val

def key_of_idx(i):
    # Square-free part of the index
    ...
```

Agent's Planning Logic (MACS-Coder)

"The problem defines a complete set based on the product of indices. Thus, two indices i, j are in the same subset iff their square-free parts are identical. I must group array values by the square-free part of their 1-based indices, not the values themselves."

Figure 8: Comparison of semantic reasoning between Direct generation and MACS-Coder for Task 3047. The Direct model incorrectly applies number theory to array values, while MACS-Coder's planning stage ensures the logic targets the indices as required by the problem constraints.

I The Multi-stage Chat-Template Design of MACS-CODER

I.1 The prompt of the STD IO Tool

The prompt of the STD IO Tool in the Deep Planning System (In Step 1)

System Prompt: You are an automated code analysis tool. Based on the following problem's input and output examples, determine which "input type" and "output format" should be used. Use the `<tool_call>` tag and output **only** this:

```
<tool_call>
input_type = {single | single-multiarr | multiple-layer1 | multiple-layer2}
output_format = {number | list | str | boolean | json}
</tool_call>
```

Classification rules:

1. **single** – Only one test case provided per execution (with variants):
   - *single-simple*: a single plain input (no extra parameters)
   - *single-multiarr*: matrix or multiple-row array, for example:

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

2. **multiple** – Multiple test cases in one run (with T first, followed by per-case data):
   - *multiple-layer1*: T lines, each a single-line test case, e.g.:

     ```
     6
     abc
     acb
     ...
     ```

   - *multiple-layer2*: multiple test cases, each spanning multiple lines, e.g.:

     ```
     4
     4
     2 2 1 2
     ...
     ```

Output format must be one of:

- *number*: an integer or floating-point number
- *list*: space-separated items
- *str*: a single string
- *boolean*: true or false
- *json*: valid JSON structure (e.g. `{"a":1,"b":2}`)

Example 1
Input:
{"input":"4 2\n1 2 3 4\n4 3 1 2\n", "output":"1 4\n"}
Expected response:

```
<tool_call>
input_type = single-multiarr
output_format = list
</tool_call>
```

Example 2
Input:
{"input":"4\n4\n2 2 1 2\n3\n0 1 2\n5\n4 3 2 3 4\n9\n9 9 9 9 9 9 9 9 9\n",
"output":"16\n2\n432\n430467210\n"}
Expected response:

```
<tool_call>
input_type = multiple-layer2
output_format = number
</tool_call>
```

Example 3
Input:
{"input": "[1, 4, 3, 8, 5]", "output": "[1, 3]" }
Expected response:

```
<tool_call>
input_type = single
output_format = list
</tool_call>
```

Example 4
Input:
{"input": "\"51230100\"", "output": "\"512301\""}
Expected response:

```
<tool_call>
input_type = single
output_format = str
</tool_call>
```

Important: Analyze the problem's intended function signature, not just the test cases. Determine the correct input_type and output_format for the function you'll implement.
Below is the problem description:
{Problem}

I.2 The prompt of the Planning Agent in the Deep Planning System

The prompt of the Planning Agent in the Deep Planning System. (In Step 2)

You are a programmer tasked with generating an appropriate plan to solve a given problem using the **{language}** programming language.

Problem

{problem}
Expected Output:
Your response must be structured as follows:

Problem Understanding

- Think about the original problem. Develop an initial understanding of the problem.

Problem Setter Perspective

Analyze the problem as if you are the problem setter:

- What core concept or algorithm is being tested?
- What kind of solution do you expect the solver to come up with?
- Why might this problem have been designed in this particular way?

Possible Solution Methods

- List **all possible algorithms or techniques** that could be applied to solve the problem.
- For each method, briefly describe:
  - Its general idea
  - Its complexity
  - Its pros and cons in the context of this problem

Chosen Method: Simplicity and Reliability

- Based on the methods listed above, choose the one that is simplest, most robust, and most likely to succeed given the problem constraints.
- Justify why this method is chosen over the others.

Recall Example Problem

Recall a relevant and distinct problem (different from the one mentioned above):

- Describe it
- Write the {language} code step-by-step to solve that problem
- Discuss the algorithm used in that example
- Generate a plan to solve that example problem

Algorithm to Solve the Original Problem

- Describe the chosen algorithm again, but now in the specific context of the original problem.
- Include tutorial-style tips:
  - How to approach this type of algorithm
  - Important pitfalls to avoid
  - Any tricks to make implementation easier

Boundary Cases to Consider

- Based on the problem description and constraints, list all boundary cases that need special handling.
- Use clear natural language to describe these boundary cases.

Plan

- Write down a detailed, step-by-step plan to solve the **original problem**.

Important Instruction:

- Strictly follow the instructions.
- Do not output any code.
- The plan must be written in natural language, not code.

I.3 The prompt of the Code Template in the Deep Planning System

The prompt of the Code Template in the Deep Planning System. (In Step 3)

Code Format Template

```
import sys, json
from typing import List, Union

# Universal input parser: automatically parse JSON, int, float, space-separated list, etc.
def parse_value(s: str) -> Union[int, float, str, list, dict, bool]:
s = s.strip()
# 1. Try JSON parsing (handles list, dict, quoted string, boolean, number)
try:
       return json.loads(s)
except json.JSONDecodeError:
       pass
# 2. Integer detection
if s.isdigit() or (s.startswith('-') and s[1:].isdigit()):
       return int(s)
# 3. Float detection
try:
       return float(s)
except ValueError:
       pass
# 4. Split on spaces *only* if the value is not enclosed in quotes
if '␣' in s and not (s.startswith('"') and s.endswith('"')):
       return [parse_value(part) for part in s.split()]
# 5. Strip surrounding quotes and keep inner spaces
if s.startswith('"') and s.endswith('"'):
       return s[1:-1]
# Fallback: return raw string
return s

# Replace this class with your own logic implementation
class Solution:
def solved_fun(self, *args):
       pass # TODO: Replace with your actual implementation

def solve(self, *args) -> Union[int, str]:
       return self.solved_fun(*args)

# Main function: handles most common competitive input/output formats
def main():
# Read all non-empty lines from stdin
```

```
lines = [parse_value(line.strip()) for line in sys.stdin if line.strip()]

# stdin template
{stdin_template}

# Python script entry point
if __name__ == "__main__":
main()
```

This Python template is designed to handle a wide variety of standard input/output formats common in programming contests. You should edit this template in three specific places when solving a new problem.

1. **`import` Section**

   If your solution requires additional Python libraries (e.g., `math`, `collections`, etc.), please add them to the `import` section at the top of the file.

   ```
   import sys, json
   from typing import List, Union
   # Add additional imports here if needed
   # e.g., from collections import defaultdict
   ```

---

2. **Write Your Solution Inside the `Solution` Class**

   Implement your logic inside the `solved_fun()` method. Rename the method according to the problem's context (e.g., `def count_pairs(self, ...)`). Make sure to update the call inside the `solve()` method accordingly.

   ```
   class Solution:
   def your_function_name(self, ...): # <-- Rename this function
          # Implement your solution here
          return ...

   def solve(self, *args) -> Union[int, str]:
          return self.your_function_name(*args) # <-- Make sure this matches
   ```

---

3. **Customize the `main()` Function's Input Parser**

   You **must retain** the following line for reading input:

   ```
   lines = [parse_value(line.strip()) for line in sys.stdin if line.strip()]
   ```

   Inspect the problem's input format carefully:

   - If the problem provides multiple lines of input, adjust how the input is grouped.
   - If you need to handle multiple test cases, add a loop around the function calls.
   - If input is structured (e.g., a matrix or multiple arrays), modify the logic to parse accordingly.

   ```
   def main():
   lines = [parse_value(line.strip()) for line in sys.stdin if line.strip()]
   # stdin template # <-- Make sure the following template matches your problem's input
       format
   ...
   # print result # <-- Make sure the following template matches your problem's output
       format
   ```

I.4 The prompt of the Coding Agent in the Deep Planning System

The prompt of the Coding Agent in the Deep Planning System. (In Step 3)

A step-by-step plan has already been created to solve this problem. Please follow the instructions below carefully:

Problem

{problem}

Planning Report

{plan}

Instructions

1. Implement the code strictly according to the provided plan.
2. Each part of the code must clearly indicate which step in the plan it corresponds to.
3. Add meaningful in-line comments in the code to explain the logic.
4. Please provide the final complete code; do not just return the `class solution`.

{prompt_of_code_template}

Output Format

Code

```
# Your code here, following the plan step by step and using the format provided
```

Important Instructions:

- Strictly follow the instructions.
- Please modify the code format according to the format of **Code Format Sample**.

I.5 Debugging Agent Prompt in the Deep Planning System

I.5.1 STD I/O Errors

Debugging Agent Prompt for STD I/O Errors in the Deep Planning System. (In Step 4)

You are given the following problem description and Python code.

Problem: {Problem}

Code:

```
    {Code}
```

Please focus only on fixing the `main()` function.
Your task is to **rewrite only the `main()` function** so that the program correctly handles input and output according to the problem description and the design of the `Solution` class.
Requirements:

- Do **not** modify any part of the `Solution` class.
- You **must retain** the following line for reading input:

```
lines = [parse_value(line.strip()) for line in sys.stdin if line.strip()]
```

- Based on the `lines` list and the problem's input format, write the rest of the `main()` logic so that it:
  1. Correctly parses the input.
  2. Calls the appropriate method(s) from `Solution`.
  3. Prints the result in the correct format.

Below is a sample input/output format you should follow:

```
{sample_io}
```

Return only the modified `main()` function code. Do not return any explanations or other parts of the script.

I.5.2 Runtime Errors

Debugging Agent Prompt for Runtime Errors in the Deep Planning System. (In Step 4)

You are a programmer who has received a solution of a problem written in **{language}**, but it fails to compile or throws an immediate runtime error. Your task is to fix the code so that it compiles and runs correctly.

Buggy Code

```
{code}
```

{test_log}
Expected Output:
Your response must be structured as follows:

Diagnosis

- Analyze the error message and buggy code.
- Identify the cause of the failure (e.g., syntax error, missing variable, etc.).
- Explain how to fix it concisely.

Modified Code

- Please keep all program comments.
- If there are errors, please also update the comments.

```
# Corrected code with concise comments on the fix.
```

Important Instructions:

- Strictly follow the instructions.
- Do not change the algorithm unless necessary to fix the compile error.
- Focus on fixing syntax or execution-level issues only.
- Do not add testing code, for example, `assert` statements in your code.

I.5.3 Wrong Answer Cases

Debugging Agent Prompt for Wrong Answer Cases in the Deep Planning System. (In Step 4)

You are a programmer who needs to fix a solution to a problem written in **{language}**. The code runs, but produces incorrect results on certain test cases.

Problem

{problem_with_planning}

Buggy Code

```
{code}
```

Test Case Failure Log

{test_log}

Expected Output:

Analyze the Plan and Test Failure

- Read the problem plan and failed test case log.
- Simulate the plan's logic using one failed test input.
- Determine what the correct output should be.
- Identify what kind of mistake likely occurred (e.g., wrong algorithm, rounding error, missing constraint).

Root Cause & Strategy

- Clearly describe the most likely root cause.
- Do not patch old logic—redesign a **better and more reliable solution** if needed.
- Mention if precision issues suggest using better tools (e.g., `decimal` instead of `float`).

Improved Plan

- Describe an improved plan or algorithm to solve the problem more accurately and reliably.

Code

- Please keep all program comments.
- If there are errors, please also update the comments.

```
# Rewritten code based on improved plan, using robust logic and tools.
```

Important Instructions:

- Strictly follow the instructions.
- Do not copy old logic.
- Be open to changing tools (e.g., decimal) or redesigning the algorithm if needed.
- Do not add testing code, for example, `assert` statements in your code.