MavenCoder: Competitive Code Generation via Model Adaptive Planning Strategies and Multi-Perspective Verification Enhancement

Zhenchun Xu[1,2,3] **Yi Cai**[1,2,3*] **Dajun Zheng**[1,2,3] **Li Yuan**[1,2,3]
Mengchen Zhao[1,2,3] **Qixiang Wang**[1,2,3] **Jiexin Wang**[1,2,3*]
[1] School of Software Engineering, South China University of Technology, Guangdong, China
[2] Key Laboratory of Big Data and intelligent Robot (South China University of Technology), Ministry of Education
[3] Joint Guangdong-Hong Kong-Macao Research Laboratory of Big Data and Robotic Intelligence, Ministry of Education
{202230484265,seqadg,seyuanli,202230484203}@mail.scut.edu.cn
{ycai,zzmc,jiexinwang}@scut.edu.cn

Abstract

With the rapid advancement of large language models (LLMs), automated code generation has made remarkable progress. Recent studies explore multi-agent collaboration and adopt planning–coding–debugging workflows to enhance performance. However, these approaches are constrained by rigid, predefined workflows that fail to flexibly adjust their plans and lack effective verification of intermediate reasoning steps. In this work, we propose **MavenCoder**, a model-adaptive and verification–enhanced framework for competition-level code generation. MavenCoder leverages adaptive assessment aligned with the model's capabilities to select planning strategies, while providing timely feedback and correction via multi-perspective verification. This adaptive problem-solving paradigm mitigates earlier limitations by enabling flexible planning and timely error correction. Compared with existing state-of-the-art approaches, MavenCoder achieves superior pass@1 results across multiple benchmarks, achieving 87.5% on LiveCodeBench, 93.9% on HumanEval+, 81.7% on MBPP+, and 46.1% on CodeContests, outperforming recent agent-based systems with improvement exceeding 3%–40% [1].

1 Introduction

Code generation aims to translate natural language specifications into executable programs (Shin et al., 2021). Recent advances in LLMs have significantly enhanced automated program synthesis (Dubey et al., 2024; Liu et al., 2024; Yang et al., 2025; Hurst et al., 2024; Comanici et al., 2025), giving rise to a growing ecosystem of AI-assisted development tools (Devi et al., 2024). Competitive programming has emerged as a widely used benchmark to systematically evaluate models' coding abilities, as it provides well-defined problem statements, strict input–output specifications, and challenging efficiency constraints (Yuen et al., 2023; Bandeira et al., 2019). This characteristic requires models to accurately analyze language requirements, select appropriate solution strategies, and generate codes that satisfy both accuracy and efficiency constraints, reflecting key competencies in real-world development.

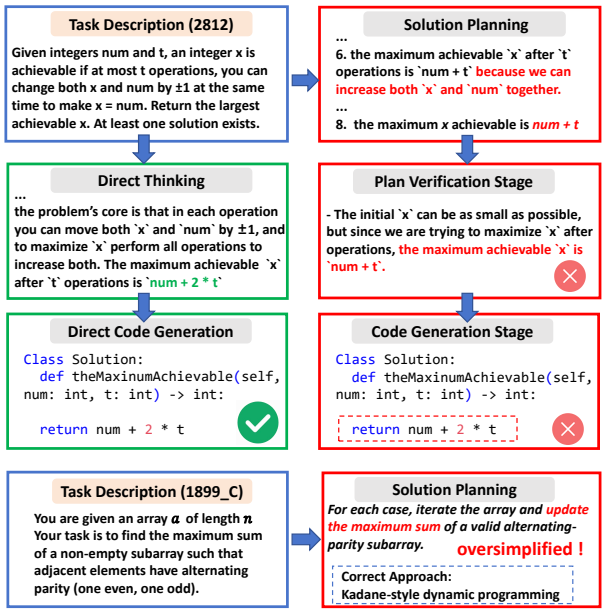

Figure 1: LCB #2812 (medium) and #1889-C (hard) using existing methods: #2812 fails due to verifier unable to correct errors, causing propagation, but direct generation solves it; #1889-C fails because simple reasoning.

Prior work on automated code generation largely builds on prompt-driven techniques, such as chain-of-thought (CoT) prompting and self-planning, which encourage models to decompose a problem into explicit intermediate steps rather than generating code in a single pass (Wei et al., 2022; Jiang et al., 2024; Zhang et al., 2023). While these methods are computationally efficient and effective for relatively simple tasks, they typically rely

*Corresponding authors.
[1]Our code is publicly available at https://github.com/XZC888/MavenCoder

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 30415–30439
July 2-7, 2026

on a single solution trajectory, making them brittle when confronted with complex problems requiring deeper reasoning. To address these limitations, more recent efforts emphasize multi-agent collaboration via structured planning–coding–debugging workflows (Islam et al., 2024; Zhang et al., 2024; Lei et al., 2025), in which a planning agent generates a solution outline, a coding agent generate executable codes following the proposed plan and finally a debugging agent analyzes failures on public tests to formulates targeted corrections.

Despite these advances, most existing methods still rely on monolithic and inflexible workflows, which suffer from two fundamental limitations. **First**, they apply a fixed planning strategy regardless of task complexity, leading to mismatched plans. As illustrated in Fig. 1, such frameworks fail on both a medium and a hard problem from LiveCodeBench (Jain et al., 2024). For problem #2812, the LLM is already capable of solving the task via its own reasoning. However, the enforced plan introduces an additional error when estimating the upper bound. It assumes the maximum achievable value is $\mathsf{num}+t$ instead of $\mathsf{num}+2t$. In contrast, problem #1889-C is a complex task requiring more advanced algorithmic design, but existing frameworks generate overly simplistic plans. **Second**, these frameworks lack effective verification mechanisms to detect and correct intermediate reasoning errors, causing early mistakes to propagate into subsequent coding stages and further degrading solution accuracy (Xue et al., 2023).

To address the above limitations, we propose **MavenCoder**, a model-adaptive and verification-enhanced framework for competitive code generation. MavenCoder consists of three core components. *First*, inspired by how human programmers adjust problem-solving strategies based on their knowledge reserves, a model-adaptive difficulty assessment module estimates problem difficulty based on confidence analysis, enabling dynamic adjustment of problem-solving strategies (Kang et al., 2025). *Second*, MavenCoder incorporates a multi-perspective verification module, inspired by step-wise validation of key steps and intermediate codes in human problem solving, to evaluate plan correctness and provide timely error correction (Lyu et al., 2023). *Finally*, a code synthesis and debugging module generates the final solution and iteratively refines it based on test feedback until all tests pass or a predefined iteration limit is reached. An overview of MavenCoder is in Fig 2.

We evaluate MavenCoder on six diverse coding benchmarks, covering both basic programming tasks and challenging real-world contest problems. The experimental results show consistent improvements across models, demonstrating its effectiveness in enhancing code generation performance. In summary, our contributions are outlined as follows:

- We introduce MavenCoder, a model-adaptive verification enhanced framework that adaptively adjusts problem-solving strategies according to model's knowledge, enabling more tailored and effective code generation.
- We leverage a verification module from various aspects to assess the correctness of solution plans, dynamically guiding corrections and mitigating error propagation.
- We validate the effectiveness and generality of our method across diverse programming benchmarks with multiple backbone LLMs, and further provide comprehensive analyses to guide future work.

2 Related Work

Automatic Program Generation: Automatic program generation has long been studied in AI (Gulwani et al., 2017). Early methods relied on symbolic reasoning or search-based techniques (Manna and Waldinger, 1971, 1980), requiring manually crafted rules and facing scalability challenges. Recent advances in large neural language models have reshaped this field, enabling direct translation of natural-language descriptions into executable programs. Data-driven neural approaches now dominate modern program synthesis, producing high-quality solutions across diverse programming tasks (Fan et al., 2023).

Large Language Models: LLMs underpin modern AI systems and are broadly categorized by design (Mienye et al., 2025; Peykani et al., 2025). General-purpose models, such as GPT (Hurst et al., 2024), are pretrained on diverse corpora and instruction-tuned for broad knowledge. Code-specialized models are trained on software repositories and synthetic code, giving them strong programming abilities for software-oriented tasks. Representative examples include Codex (Chen, 2021), CodeGen (Nijkamp et al., 2022), CodeLLaMA (Roziere et al., 2023), StarCoder (Li et al., 2023b), MoTCoder (Li et al., 2023a), DeepSeek Coder (Daya Guo, 2024), and WizardCoder (Luo et al., 2023).

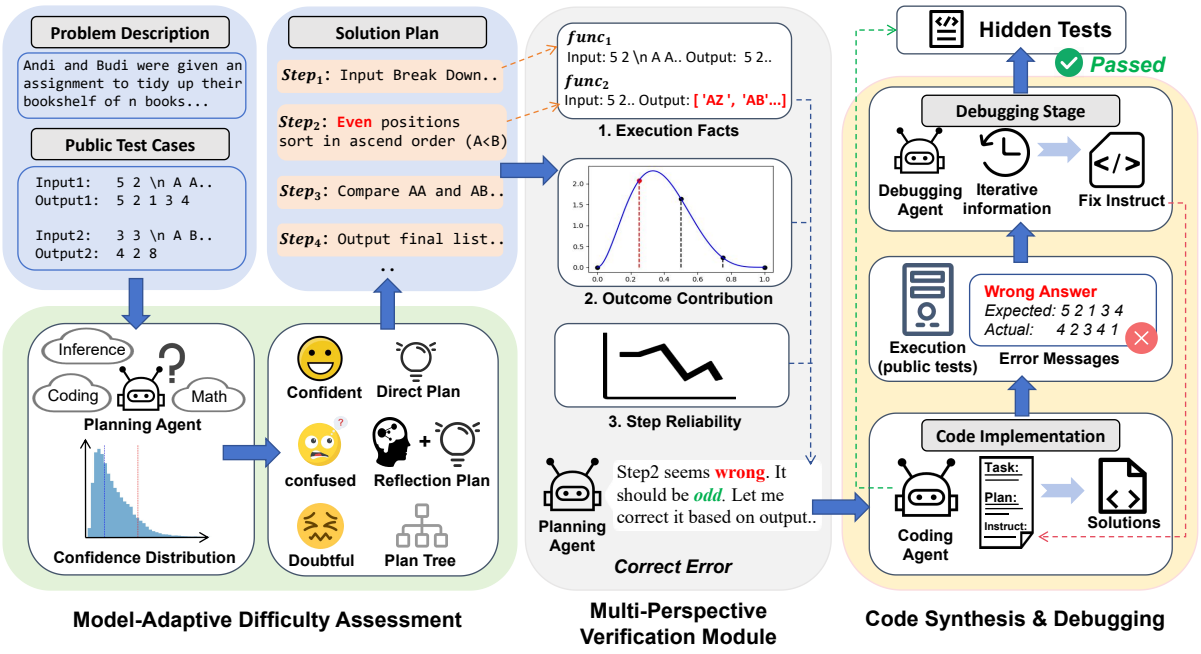

Figure 2: The overview of MavenCoder, which integrates adaptive difficulty assessment, plan generation tailored to the estimated complexity, and iterative refinement through code-generation and debugging phases.

3 Methodology

3.1 Problem Modeling

A programming task is defined as a problem instance $\mathcal{P} = (c, \mathcal{T})$, where c is detailed problem description and $\mathcal{T}$ denotes according tests, comprising two subsets:

$$\mathcal{T} = \mathcal{T}_{\text{pub}} \cup \mathcal{T}_{\text{priv}} = \{(I_j, O_j)\}_{j=1}^{m_p} \cup \{(I_k, O_k)\}_{k=1}^{m_h}$$

The first set $\mathcal{T}_{\text{pub}}$ represents public examples that provide observable signals to help coding and debugging. In contrast, the second set $\mathcal{T}_{\text{priv}}$ is inaccessible tests throughout entire procedure and serves solely as the final correctness oracle. Each pair (I, O) specifies an input instance and its expected output. Given c along with $\mathcal{T}_{\text{pub}}$, models are required to generate a candidate solution $S = [t_1, t_2, \ldots, t_N]$, where t_i denotes the i-th token. A candidate program S is deemed correct if its execution reproduces expected outputs on all hidden tests, satisfies $\forall (I, O) \in \mathcal{T}_{\text{priv}},\ S(I) = O$.

3.2 Model-Adaptive Difficulty Assessment

LLMs exhibit heterogeneous performance due to differences in training data, the perceived difficulty of the same programming problem can vary substantially across models. To capture such model-specific differences, we adopt a token-level confidence measure. For each generated token t_i, the confidence is defined as the negative mean of the top-k log-probabilities:

$$c_i = -\frac{1}{k} \sum_{j=1}^{k} \log p_i(j)$$

where higher c_i reflects increased certainty in the generated response.

Unlike prior work (Fu et al., 2025) that aggregates confidence over all tokens, we distinguish **template tokens** [2], arising from stylistic conventions such as Markdown symbols (e.g., #, **), empty-line padding, or introductory phrases (e.g., *"Let's analyze the problem"*), from **inference tokens**, which reflect substantive reasoning. Template tokens often receive artificially high confidence due to instruction-tuning, potentially distorting difficulty estimation. To address this, we detect template tokens using heuristic patterns, label their confidence score as ∞, and compute reasoning confidence via sliding-window aggregation, skipping any window containing ∞ values, as template tokens tend to cluster and bias nearby confidence values:

$$R_i = \frac{1}{|R_i|} \sum_{c_j \in R_i} c_j, \quad c_j \neq \infty$$

[2] Detailed analysis of the impact of template tokens is provided in Appendix C.2.

The sequence R_i traces the evolution of certainty along the reasoning trajectory. We further apply k-means clustering (Arthur and Vassilvitskii, 2007) to partition confidence levels into **confident** and **unconfident** groups. Based on the cluster centers θ_{low} and θ_{high}, tokens are categorized accordingly, and the proportions p_{low} and p_{high} are computed. The task difficulty from models' perspective is considered low if $p_{\text{high}} \geq \theta_1$, high if $p_{\text{low}} \geq \theta_2$, and moderate otherwise, where θ_1 and θ_2 are hyperparameters controlling threshold selection. This procedure ensures that assigned difficulty reflects the model's intrinsic reasoning confidence. For models without access to internal states, difficulty is instead determined through instruction-based classification.

3.3 Adaptive Planning Strategies

Considering the models' own assessment of problem difficulty, planning agent employs different approach strategies. Given high-confidence problems (**Easy**), direct plan generation is applied to reduce errors from overthinking. For moderate confidence (**Medium**), models first engage in self-reflective (Shinn et al., 2023) thinking in example-based reasoning, relevant algorithmic knowledge, and subsequently produces an intermediate solution plan.

For high-uncertainty tasks (**Hard**), the planning agent employs a tree-based decomposition (Press et al., 2023) guided by iterative deepening search. The problem context c forms the root N_0 of a plan tree T, with intermediate nodes N_i as subproblems and leaves representing directly solvable tasks. T is constructed level-wise, decomposing nodes hierarchically until subproblems are simple or the maximum depth D_{max} is reached. To encourage diversity, a repository $\mathcal{D} = \{(P_i, e_i, s_i)\}_{i=1}^{N}$ stores previous subproblems, embeddings, and solutions; new subproblems with embedding similarity above θ_{sim} are treated as redundant and excluded. Retrieved subproblems guide solution construction when all children are repetitive. Finally, it is linearized into a structured representation, preserving hierarchical features through indentation and subheadings.

3.4 Multi-Perspective Verification Module

To detect potential errors and ambiguities, we employ a verification module to evaluate the correctness of generated plan $\mathcal{A}$, which typically consists of a sequence of steps, $\mathcal{A} = \{a_i\}_{i=1}^{S}$. Specifically, we consider three aspects inspired by real coding practices: step reliability, factual correctness and contribution to final answer.

1. Step Reliability $r(a_i)$**.** Following the previous confidence definition, we first compute the mean confidence of each reasoning step:

$$\bar{c}_i = \frac{1}{|a_i|} \sum_{t \in a_i} c_t, \quad c_t \neq \infty$$

where $\bar{c}_i$ denotes the step-wise averages. To express the model's relative confidence across steps in a probabilistic manner, we normalize these values using a softmax:

$$r(a_i) = \frac{\exp(\bar{c}_i)}{\sum_{j=1}^{S} \exp(\bar{c}_j)}$$

This normalization maps step-level scores onto a probability simplex, ensuring that relative differences in confidence are preserved while providing a calibrated measure of the model's preference over reasoning steps.

2. Factual correctness $f(a_i)$**.** To address the challenge of associating reasoning steps with executable semantics, we employ a modular generation method: given current step a_i, the model is required to generate a corresponding modular function func_i, explicitly specifying its input–output signature, intermediate variables and expected behavior. This process yields a self-consistent set of step-level functions that align with the broader task semantics. Each public test $t = (I_j, O_j) \in \mathcal{T}_{\text{pub}}$ is executed on func_i. The execution traces are then dynamically incorporated back into the prompt, requiring models to evaluates the factual behavior of func_i against its own step-level specification. Formally, the execution fact correctness of step a_i is quantified as:

$$f(a_i) = \frac{1}{m_p} \sum_{j=1}^{m_p} \mathbf{1}_{\{\text{func}_i(I_j) = O_j\}}$$

The execution feedback thus assesses each step with respect to its model-declared semantics, using the injected runtime evidence as an objective basis for verification. This yields an interpretable, data-grounded signal that reflects how the step behaves under concrete execution conditions.

3. Outcome Contribution $w(a_i)$**.** Unlike mathematical questions, solution traces for programming tasks exhibit a strong structural regularity. Specifically, the initial and final steps are typically devoted to input parsing and output formatting, which are

largely mechanical and are handled reliably by current models, rarely contributing to errors. In contrast, the core reasoning and algorithmic logic predominantly emerge in the intermediate steps, where mistakes are more frequent and contributions to the final outcome are substantially higher.

To explicitly model this position-dependent contribution bias, we map step index i to the unit interval via $x_i = (i-1)/(S-1)$ and parameterize the relative contribution by a Beta density:

$$w(a_i) = 1 - \frac{\mathrm{Beta}(x_i;\, \alpha, \beta)}{\sum_{j=1}^{S} \mathrm{Beta}(x_i;\, \alpha, \beta)}$$

where α and β are the hyperparameters controlling the shape of the positional prior. By appropriately tuning them, the resulting weight distribution exhibits a left-skewed tendency, reflecting two considerations: the initial and terminal steps are largely devoted to I/O handling and are handled reliably, while errors introduced in the early reasoning phase tend to propagate downstream, amplifying their impact on later steps. Consequently, greater emphasis is placed on early-to-mid reasoning steps.

Comprehensive Step Value $v(a_i)$. Finally, the three factors are combined multiplicatively to form a comprehensive step score:

$$v(a_i) = r(a_i) \cdot f(a_i) \cdot w(a_i)$$

which considers reliability, factual validity, and positional contribution for each step. A step-specific threshold is then defined asθ_{step}:

$$\theta_{\text{step}} = \tau \cdot \overline{C} \cdot w_{\min}$$

where τ controls the minimum acceptable level of correctness on $\mathcal{T}_{\text{pub}}$, $\overline{C}$ denotes the average step-level confidence, and $W_{\min}$ enforces a lower bound corresponding to the least significant step. If $v(a_i) < \theta_{\text{step}}$, the verification module flags a_i as suspect, guiding the model to localize and revise the earliest unreliable reasoning step based on execution traces. This review-and-revision procedure is repeated r_{valid} rounds for error detection and correction.

3.5 Code Synthesis and Debugging

The coding agent produces an initial code S_0 from c and verified Plan with code generation instruct. S_0 is then evaluated on $\mathcal{T}_{\text{pub}}$; if all tests pass, it proceeds to $\mathcal{T}_{\text{priv}}$ for final evaluation. Otherwise, the debugging agent generates a repair instruction I_0 based on error information Error_0 to summarize fault location, cause analysis, and suggested fixes. The coding agent then generates an updated solution S_1 conditioned on I_0. In iteration i, previous solutions and error information are incorporated, enabling more effective fault identification and correction. Formally, solution S_{i+1} relies on debugging iterative information:

$$P(S_{i+1} \mid I_i) = P\Big(S_{i+1} \,\Big|\, \{S_j, \text{Error}_j\}_{j=0}^{i}\Big)$$

This coding-and-repair loop is repeated for r_{debug} times and ensures corrections are informed by prior errors, improving overall reliability and convergence of the synthesized program.

4 Experiments

4.1 Benchmarks

We evaluate MavenCoder on six widely used public benchmarks covering both basic programming tasks and competitive programming problems. Basic-level benchmarks include **HumanEval (HE)** (Chen, 2021) , **MBPP** (Austin et al., 2021), and their **Evalplus** (Liu et al., 2023), all focusing on simple function-level code generation, while Evalplus extends the original datasets with stricter tests, offering a more reliable measure of a model's true capability. As these datasets lack explicit public tests, we extract all visible assertions from their docstrings. Competition-level benchmarks include test set of **CodeContests (CC)** (Li et al., 2022) and **LiveCodeBench (LCB)** version *V1*, which feature complex reasoning and real-world problems.

4.2 Baselines

We compare MavenCoder with five categories of existing approaches. **Prompt Engineering Technologies** guide models using structured reasoning techniques such as Chain-of-Thought (CoT) and Self-Planning; we also include Direct Prompting with both random and greedy decoding strategies (Holtzman et al., 2019; Sutskever et al., 2014; Kojima et al., 2022). **Confidence-based Methods** guide sampling with uncertainty estimates: HonestCoder (Li et al., 2024) selects high-confidence programs using multi-modal similarity, and we additionally use Self-Ask to score candidate programs and choose the highest-confidence outputs. **Repair-based Methods** iteratively refine programs through execution feedback, including LDB, which verifies decomposed program blocks using intermediate

Table 1: pass@1 performance (%) across all methods. Bold denotes the best result for each benchmark, underlining indicates the second-best result, and green highlights improvements over the random baseline. '–' indicates unavailable results due to reproduction issues.

Method	GPT-4.1-nano						GPT-4o-mini						Qwen3-Coder+					
	HE	HE+	MBPP	MBPP+	LCB	CC	HE	HE+	MBPP	MBPP+	LCB	CC	HE	HE+	MBPP	MBPP+	LCB	CC
Direct (Random)	89.5	84.1	85.1	71.0	48.5	12.7	87.8	83.4	86.7	73.1	43.2	10.5	96.5	92.2	94.1	79.7	76.5	24.8
Direct (Greedy)	89.0	82.9	86.8	72.8	48.3	11.5	88.4	83.5	85.2	72.0	43.8	10.9	97.0	93.3	94.4	80.2	76.8	23.6
CoT	79.3	75.6	80.2	68.8	36.3	9.1	75.6	73.2	86.5	73.3	40.8	9.1	82.9	80.5	92.1	78.8	74.0	15.2
Self-Planning	86.6	81.1	86.0	73.0	33.5	6.1	84.8	81.1	86.0	70.4	41.3	9.7	92.7	87.2	93.1	78.6	81.5	29.1
HonestCoder	91.0	85.5	85.5	72.1	50.8	12.4	87.7	83.5	86.7	73.4	43.9	10.3	97.2	92.1	90.9	76.8	77.0	26.1
Self-Ask	91.7	86.1	86.6	72.3	50.9	12.4	87.8	83.4	86.9	73.0	45.0	10.5	96.6	91.7	90.7	76.6	76.8	25.5
SR	91.9	87.3	91.7	76.6	54.0	16.7	91.5	85.4	92.1	76.5	47.3	11.6	98.1	91.2	94.2	79.6	77.5	25.3
LDB	90.2	86.0	94.4	77.5	55.8	20.0	91.5	86.6	92.6	76.5	49.5	11.5	98.2	94.5	98.9	81.7	-	-
MapCoder	87.8	81.1	93.1	75.4	35.5	22.4	92.1	84.1	90.2	73.8	50.5	21.8	97.6	92.7	98.1	81.0	-	37.6
PairCoder	93.9	87.8	89.7	72.2	57.5	26.7	96.3	87.2	93.9	75.1	51.0	24.8	98.8	94.5	92.6	77.5	79.3	44.2
LPW	94.5	86.0	94.7	76.2	57.8	24.2	90.9	81.1	94.4	73.8	41.0	18.2	**99.4**	90.2	98.1	81.7	78.8	38.8
ToT	92.7	87.8	90.2	76.2	54.0	7.3	84.8	79.3	89.7	73.5	41.5	9.7	95.7	90.2	95.5	81.7	83.5	29.7
CodeTree	93.9	88.4	94.7	77.2	60.3	18.2	90.9	84.1	**96.6**	76.2	54.8	17.6	96.3	90.9	97.9	83.3	85.5	37.6
MavenCoder	**97.6** ↑ 9.1%	**90.2** ↑ 7.3%	**97.1** ↑ 14.1%	**81.0** ↑ 14.1%	**70.0** ↑ 44.3%	**32.7** ↑ 157.7%	**98.2** ↑ 11.8%	**89.0** ↑ 6.7%	95.0 ↑ 9.6%	**79.4** ↑ 8.6%	**56.8** ↑ 31.5%	**25.5** ↑ 142.9%	98.8 ↑ 2.4%	**96.3** ↑ 4.4%	**98.9** ↑ 5.1%	81.7 ↑ 2.5%	**87.5** ↑ 14.4%	**46.1** ↑ 85.9%

states (Zhong et al., 2024), and Self-Repair (SR), which diagnoses and fixes errors through iterative runs (Rushing and Nanda, 2024). **Agent-based Approaches** employ multi-agent coordination for planning and coding: MapCoder executes a sequential plan–implement–debug workflow; LPW enhances planning through example-based reasoning and improves coding via model-derived explanations; PairCoder simulates pair programming with a Navigator for planning and analysis and a Driver for implementation. **Tree-based Exploration Methods** expand the search space by exploring multiple solution branches, as exemplified by Tree-of-Thought (ToT) frameworks (Yao et al., 2023) and CodeTree (Li et al., 2025), which rank and refine partial programs within a unified tree structure.

4.3 Experimental Setup

We evaluate both general-purpose and code-specialized models, covering closed-source LLMs, GPT-4o-mini and GPT-4.1-nano (OpenAI et al., 2024), open-source models Qwen3-Coder+ (Yang et al., 2025). All methods [3] are assessed using the standard pass@1 metric (Chen, 2021), and best result is reported over two runs.

4.4 Pass@1 Performance Comparison

Table 1 reports the Pass@1 performance across different methods. On competitive benchmarks, MavenCoder surpasses all baselines, and achieves comparable or superior results on basic ones. Compared with recent strong baselines like LPW, PairCoder and CodeTree, MavenCoder achieves dramatic gains: **44.3%** improvement on LCB and **157.7%** on CC for GPT-4.1-nano, **142.9%** for GPT-4o-mini, and **85.9%** for Qwen3-Coder+. These improvements largely stem from capability-aligned adaptive planning, which better exploits models' internal knowledge. For simpler tasks, Our method delivers limited improvements; we attribute this to the prevalence of simple reasoning patterns, where the model's direct solutions are sufficiently correct despite involving fewer deliberation steps. More detailed results are shown in Appendix E.

[3] Detailed settings for each method are provided in Appendix B. For MavenCoder, we follow prior work or adopt empirically validated configurations (see Appendix D).

Table 2: Ablation results (pass@1, %) on CC and HE. Red numbers indicate performance drops relative to MavenCoder. **Type** treats all tasks as the same level.

Ablation Setting	Type	GPT-4.1-nano CC	GPT-4.1-nano HE	GPT-4o-mini CC	GPT-4o-mini HE
w/o adaptive strategy	Easy	27.8 ↓ 4.9	95.7 ↓ 1.9	21.2 ↓ 4.3	95.1 ↓ 3.1
	Medium	27.3 ↓ 5.4	95.1 ↓ 2.5	18.8 ↓ 6.7	93.9 ↓ 4.3
	Hard	29.7 ↓ 3.0	95.7 ↓ 1.9	18.8 ↓ 6.7	96.3 ↓ 1.9
w/o planning stage	–	26.7 ↓ 6.0	95.1 ↓ 2.5	18.2 ↓ 7.3	94.5 ↓ 3.7
w/o verification module	w/o $r(a_i)$	26.7 ↓ 6.0	96.3 ↓ 1.3	24.2 ↓ 1.3	95.7 ↓ 2.5
	w/o $f(a_i)$	26.1 ↓ 6.6	95.7 ↓ 1.9	24.8 ↓ 0.7	93.3 ↓ 4.9
	w/o $w(a_i)$	24.8 ↓ 7.9	95.7 ↓ 1.9	23.0 ↓ 2.5	90.9 ↓ 7.3
w/o debugging stage	–	20.6 ↓ 12.1	95.1 ↓ 2.5	15.2 ↓ 10.3	91.5 ↓ 6.7

4.5 Ablation Study

To quantify the contribution of each component in MavenCoder, we conducted ablation studies (Table 2). For challenging problems, removing the debugging modules leads to substantial performance drops, highlighting its critical roles in detecting execution errors and providing corrective guidance. Performance also degrades when the planning mod-

Table 3: Comparison of different confidence estimation signals across all benchmarks.

Benchmark	GPT-4o-mini			GPT-4.1-nano		
	Mean	*Entropy*	*Prompt*	*Mean*	*Entropy*	*Prompt*
HE	**98.2**	95.1	96.3	**97.6**	95.1	95.7
HE+	89.0	**90.9**	90.2	90.2	90.2	**90.9**
MBPP	95.0	**95.8**	95.2	**97.1**	96.0	96.3
MBPP+	**79.4**	76.2	75.9	**81.0**	78.0	77.5
LCB	56.8	57.8	**58.5**	**70.0**	65.8	67.5
CC	**25.5**	21.8	23.0	**32.7**	29.7	29.1

ule is removed, suggesting that explicit planning is necessary to solve problems. The adaptive mechanism is essential as well: when a uniform workflow is enforced and all tasks are treated as simple, medium, or hard, performance degrades consistently, demonstrating the necessity of dynamically adjusting planning strategy according to model abilities and task difficulty. Removing the verification module also leads to performance degradation, with factual correctness contributing the most, highlighting the importance of input–output-based validation to ensure step accuracy. Overall, each module contributes meaningfully, with debug and planning being the most influential.

4.6 Confidence Estimation Analysis

We investigate how different confidence estimation signals affect adaptive difficulty stratification for model evaluation. Specifically, we compare three representative approaches: (i) the mean negative log-probability of tokens, as defined above. (ii) entropy-based uncertainty estimation, given by

$$h_i = -\sum_{j=1}^{k} p_i(j) \log p_i(j)$$

(iii) direct difficulty classification via prompt.
As shown in Table 3, incorporating confidence signals into adaptive difficulty modeling consistently improves performance across benchmarks, outperforming strong baselines. Despite difference in performance improvement, all signals yield similar gains, suggesting that the benefit stems from leveraging the model's intrinsic self-assessment rather than any specific metric.

4.7 Model-Adaptive Difficulty Distribution

We analyze predicted difficulty distributions on LCB, whose balanced labels align with our categories. Fig. 3 shows notable differences across models. GPT-4o-mini, a large general-purpose model, produces relatively balanced predictions

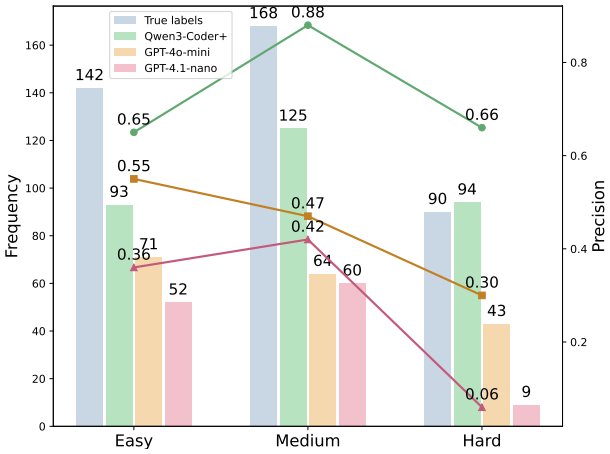

Figure 3: Adaptive Difficulty Prediction Distribution on LCB. The bars indicate prediction frequency for each difficulty level, while the line denotes prediction accuracy.

for easy and medium problems but underestimates the hardest tasks. GPT-4.1-nano, smaller and conservative, rarely predicts hard problems, lowering overall accuracy. In contrast, the code-specialized Qwen3-Coder+ predicts a larger proportion of problems across all levels with higher accuracy. These differences indicate that models assess difficulty based on their knowledge, favoring tasks they handle confidently [4]. Importantly, this subjective assessment reflects each model's internal capabilities and preferences, so alignment with truth difficulty labels is **less critical** than capturing model's behavior in our study.

Metric	Self-Planning	LPW	MavenCoder
Avg Score	4.0	3.8	4.4

Table 4: Plan Quality Comparison (Qwen3-Coder+).

4.8 Human Evaluation

To further assess plan quality, we randomly sampled 20 tasks per benchmark and asked three experienced competitive programmers to independently evaluate the generated plans. Each plan was rated on a 1–5 scale based on factual correctness, clarity, and reasoning completeness. As shown in Table 4, our method consistently achieves higher scores across all benchmarks. Moreover, the low standard deviations (1.07, 1.31, and 1.32) indicate strong inter-rater agreement rather than evaluator

[4]We present an illustrative example in Appendix Fig 29, showing that different models perceive the difficulty of the same task differently.

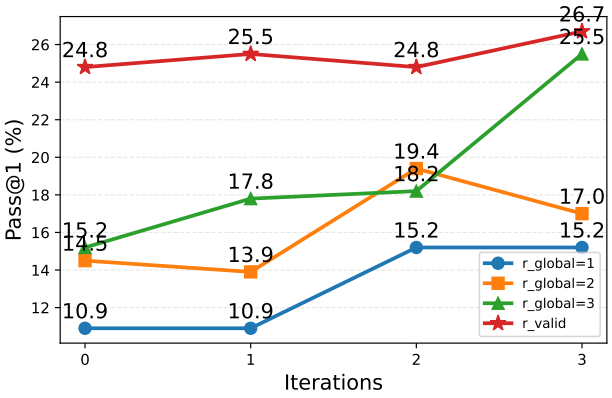

Figure 4: Performance Variation Across Iteration Frequency using GPT-4o-mini on CC.

bias. These results demonstrate that combining adaptive planning with multi-stage verification produces plans that are more accurate, coherent, and conducive to deep reasoning [5].

4.9 Effect of Iteration Rounds

We analyze the impact of iterative rounds on model performance. As shown in Fig. 4, increasing iterations generally improves accuracy. Global iterations allow exploration of diverse solution trajectories, verification iterations guide the model in correcting potential errors, and debugging iterations refine residual logical flaws. Each additional round contributes cumulatively to performance gains, with global iterations particularly effective due to the exploration of multiple reasoning paths. This highlights the importance of iterative refinement for robust, high-quality solutions.

$\theta_1 \backslash \theta_2$	35	45	55	35	45	55
5	21.2 160/0/5	23.0 157/1/7	21.2 158/5/2	87.2 151/0/13	**89.0** **133/9/22**	87.8 155/6/3
15	20.0 33/31/101	**25.5** **30/117/17**	23.0 32/132/1	84.1 29/26/109	86.6 38/101/25	87.2 37/122/5
25	22.4 0/64/101	20.6 0/148/17	17.0 0/165/0	86.0 1/54/109	87.2 0/140/24	87.8 2/155/7

Table 5: Comparison of different θ_1 and θ_2 (left: CC, right: HE+). Top row shows pass@1 results; bottom row shows task distribution across easy/medium/hard.

4.10 Effect of Confidence Threshold

To investigate what difficulty distribution lead to optimal performance, we experiment with different combinations of confidence thresholds θ_1 and θ_2 using GPT-4o-mini. As shown in Table 5, smaller θ_1 values favor easy tasks, whereas smaller θ_2 values bias toward hard tasks. For competitive problems, the best performance is achieved when medium-difficulty tasks dominate, with moderate proportions of easy and hard instances. For simpler problems, performance remains stable when easy or medium tasks are prevalent but declines if the allocation is overly skewed toward hard tasks, suggesting that excessive reasoning depth can be detrimental for simpler problem types.

[5] We provide a representative example in Appendix G.

Table 6: Time and token consumption comparison using Qwen3-Coder+. '–' indicates the repository does not support or unavailable due to reproduction issues.

Approach	MBPP+		CC		LCB	
	Time	Token	Time	Token	Time	Token
LPW	2h	22.6M	9h	97.9M	12h	200+M
MapCoder	5h10m	**2.9M**	10h	10.2M	–	–
PairCoder	5h30m	–	4h53m	–	7h35m	–
MavenCoder	**1h17m**	8.2M	**4h30m**	**8.1M**	**7h5m**	**5.8M**

4.11 Token and Time Consumption

To evaluate practical feasibility, we compare runtime and token usage across agent-based methods using Qwen3-Coder+ as backbone, under a unified budget of 9 iteration rounds and a concurrency of 6. As summarized in Table 6, MavenCoder consistently achieves a favorable balance between efficiency and effectiveness, exhibiting shorter end-to-end runtimes in most settings while maintaining low token consumption. This advantage stems from the dynamic adjustment of planning strategies, which improves the overall accuracy of generated plans and consequently reduces the need for extensive iterative refinement. Overall, the results indicate that MavenCoder maintains controllable cost, making it suitable for large and time-sensitive real-world development scenarios.

5 Conclusion

In this work, we proposed **MavenCoder**, a model-adaptive and verification-enhanced framework for competition-level code generation. MavenCoder adaptively aligns planning strategies with the model's capabilities and employs a verification module from different aspects to assess solution plans, overcoming the rigidity and weak self-correction of existing workflows. Extensive experiments across major text-to-code benchmarks show consistent pass@1 improvements over superior methods, demonstrating the effectiveness of

adaptive reasoning and precise verification for robust program synthesis.

Future work will extend MavenCoder to broader software development scenarios, such as repository-level code generation, to systematically evaluate its effectiveness in handling cross-file dependencies and reasoning over large-scale code contexts.

Limitations

Despite its effectiveness, our approach has several limitations. First, we currently evaluate our approach only on LLMs, as they typically exhibit stronger instruction-following capabilities and can reliably produce outputs in the expected formats. Experiments on locally deployed medium-scale models are left for future work. Second, some closed-source or reasoning-focused LLMs do not expose token-level log-probabilities, which prevents confidence estimation and limits applicability. Finally, while the method significantly improves performance on medium and hard problems, its benefits on simple tasks are limited, as strong base models often already succeed without additional planning or verification.

Acknowledgments

This research is supported by the National Natural Science Foundation of China (62402185), the Fundamental Research Funds for the Central Universities" (2025ZYGXZR064), the Science and Technology Planning Project of Guangdong Province (2025B0101120003), the Guangdong Provincial Fund for Basic and Applied Basic Research—Regional Joint Fund Project (Key Project) (2023B1515120078), the Guangdong Provincial Natural Science Foundation for Outstanding Youth Team Project (2024B1515040010), the Fundamental Research Funds for the Central Universities, South China University of Technology (x2rjD2250190).

References

David Arthur and Sergei Vassilvitskii. 2007. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics.

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

Ian Nery Bandeira, Thiago Veras Machado, Vitor F Dullens, and Edna Dias Canedo. 2019. Competitive programming: A teaching methodology analysis applied to first-year programming classes. In *2019 IEEE Frontiers in Education Conference (FIE)*, pages 1–8. IEEE.

Mark Chen. 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, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, 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*.

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

Dr S Rama Devi, Ommi U CH BhagyaSri, R Sravanthi, SL Chaitrika, MN Priyanka, M Swarna, and M Srilekha. 2024. Ai-enhanced cursor navigator. *R. and Chaitrika, SL and Priyanka, MN and Swarna, M. and Srilekha, M., AI-Enhanced Cursor Navigator (May 10, 2024)*.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, and 1 others. 2024. The llama 3 herd of models. *arXiv e-prints*, pages arXiv–2407.

Ahmed El-Kishky, Alexander Wei, Andre Saraiva, Borys Minaiev, Daniel Selsam, David Dohan, Francis Song, Hunter Lightman, Ignasi Clavera, Jakub Pachocki, and 1 others. 2025. Competitive programming with large reasoning models. *arXiv preprint arXiv:2502.06807*.

Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pages 31–53. IEEE.

Yichao Fu, Xuewei Wang, Yuandong Tian, and Jiawei Zhao. 2025. Deep think with confidence. *arXiv preprint arXiv:2508.15260*.

Team Glm, Aohan Zeng, Bin Xu, Bowen Wang, Chenhui Zhang, Da Yin, Dan Zhang, Diego Rojas, Guanyu Feng, Hanlin Zhao, and 1 others. 2024. Chatglm: A family of large language models from glm-130b to glm-4 all tools. *arXiv preprint arXiv:2406.12793*.

Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, and 1 others. 2017. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2019. The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751*.

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

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, and 1 others. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.

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

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

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.

Zhewei Kang, Xuandong Zhao, and Dawn Song. 2025. Scalable best-of-n selection for large language models via self-certainty. *arXiv preprint arXiv:2502.18581*.

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

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.

Jia Li, Yuqi Zhu, Yongmin Li, Ge Li, and Zhi Jin. 2024. Showing llm-generated code selectively based on confidence of llms. *arXiv preprint arXiv:2410.03234*.

Jierui Li, Hung Le, Yingbo Zhou, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. 2025. Codetree: Agent-guided tree search for code generation with large language models. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 3711–3726.

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

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, and 1 others. 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, and 7 others. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

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

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

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*.

Qing Lyu, Shreya Havaldar, Adam Stein, Li Zhang, Delip Rao, Eric Wong, Marianna Apidianaki, and Chris Callison-Burch. 2023. Faithful chain-of-thought reasoning. In *The 13th International Joint Conference on Natural Language Processing and the 3rd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics (IJCNLP-AACL 2023)*.

Zohar Manna and Richard Waldinger. 1980. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):90–121.

Zohar Manna and Richard J Waldinger. 1971. Toward automatic program synthesis. *Communications of the ACM*, 14(3):151–165.

Ibomoiye Domor Mienye, Nobert Jere, George Obaido, Oyindamola Omolara Ogunruku, Ebenezer

Esenogho, and Cameron Modisane. 2025. Large language models: an overview of foundational architectures, recent trends, and a new taxonomy. *Discover Applied Sciences*, 7(9):1027.

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

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, and 262 others. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Pejman Peykani, Fatemeh Ramezanlou, Cristina Tanasescu, and Sanly Ghanidel. 2025. Large language models: A structured taxonomy and review of challenges, limitations, solutions, and future directions. *Applied Sciences*, 15(14):8103.

Ofir Press, Muru Zhang, Sewon Min, Ludwig Schmidt, Noah A Smith, and Mike Lewis. 2023. Measuring and narrowing the compositionality gap in language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 5687–5711.

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

Cody Rushing and Neel Nanda. 2024. Explorations of self-repair in language models. *arXiv preprint arXiv:2402.15390*.

Jiho Shin, Jaechang Nam, and 1 others. 2021. A survey of automatic code generation from natural language. *Journal of Information Processing Systems*, 17(3):537–555.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to sequence learning with neural networks. *Preprint*, arXiv:1409.3215.

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

Tianci Xue, Ziqi Wang, Zhenhailong Wang, Chi Han, Pengfei Yu, and Heng Ji. 2023. Rcot: Detecting and rectifying factual inconsistency in reasoning by reversing chain-of-thought. *Preprint*, arXiv:2305.11499.

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

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

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*.

Kevin KF Yuen, Dennis YW Liu, and Hong Va Leong. 2023. Competitive programming in computational thinking and problem solving education. *Computer Applications in Engineering Education*, 31(4):850–866.

Huan Zhang, Wei Cheng, Yuhan Wu, and Wei Hu. 2024. A pair programming framework for code generation via multi-plan exploration and feedback-driven refinement. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1319–1331.

Jian Zhang, Zhangqi Wang, Haiping Zhu, Kangda Cheng, Kai He, Bo Li, Qika Lin, Jun Liu, and Erik Cambria. 2026a. Mars: Multi-agent adaptive reasoning with socratic guidance for automated prompt optimization. *Proceedings of the AAAI Conference on Artificial Intelligence*, 40(19):16307–16315.

Jian Zhang, Zhiyuan Wang, Zhangqi Wang, Fangzhi Xu, Qika Lin, Lingling Zhang, Rui Mao, Erik Cambria, and Jun Liu. 2026b. Maps: Multi-agent personality shaping for collaborative reasoning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 40(19):16316–16324.

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*.

Appendix

A Additional Related Work

A.1 Agent-based LLM Systems

Recent advances in LLMs have led to the emergence of agent-based systems, which extend pure text generation into goal-directed, interactive problem solving. Unlike conventional prompting paradigms, LLM agents are equipped with capabilities such as iterative reasoning, tool use, environment interaction, and self-reflection, enabling them to decompose complex tasks and adapt their behavior based on intermediate feedback.

A representative paradigm is the ReAct framework (Yao et al., 2022), which interleaves reasoning and acting by allowing models to generate both intermediate thoughts and executable actions. This design facilitates dynamic interaction with external tools and has been widely adopted in subsequent agent systems.

Agent-based approaches have been successfully applied across diverse domains, including code generation (Huang et al., 2023), fundamental science task solving (Zhang et al., 2026b), and domain-specific problem solving (Zhang et al., 2026a). These systems demonstrate improved robustness by exposing intermediate reasoning processes and enabling verification or correction at different stages. Our work aligns with this paradigm by incorporating adaptive planning and verification mechanisms within an agentic framework, aiming to enhance reliability in program synthesis under varying task difficulty.

B Implementation Details

In this section, we present the detailed setup of our experiments. We retain the original best-performing settings and modify only the parameters explicitly noted below for each method, ensuring reproducibility and fair comparisons across baselines.

All evaluations use standard official frameworks: **EvalPlus** for basic benchmarks and **LiveCodeBench** for competitive ones with 6 seconds timeout, ensuring reliable and rigorous assessment.

B.1 MavenCoder

Following prior work, we compute confidence scores using the top-5 token log-probabilities and apply a sliding window of size 5 to improve stability. Based on these scores, we set the confidence thresholds to $\theta_1 = 15\%$ (reduced to 5% for simple tasks) and $\theta_2 = 45\%$, which bias the model toward generating plans of moderate difficulty. As shown in Table 5, this preference yields better overall performance, while simpler threshold settings are more effective for basic problems. To control the Beta distribution used for plan weighting, we set the shape parameters $\alpha = 3$ and $\beta = 5$, reflecting the observed average plan length of 8 steps. We further set $\tau = 0.5$ in θ_{step}, requiring at least half of the public tests to be passed. For iteration settings, we use $r_{\text{valid}} = 1$ during plan validation, as this stage requires strong instruction-following and higher repetitions were found to be error-prone in practice. We also set $r_{\text{global}} = 3$ and $r_{\text{debug}} = 3$. The maximum plan-tree depth is set to $D_{\max} = 5$, with an embedding similarity threshold of $\theta_{\text{sim}} = 0.7$. Except for the plan-generation stage, where we use temperature 0.8 and top-p 0.95 to encourage diverse plan exploration—all subsequent stages adopt greedy decoding (temperature 0) to ensure deterministic and accurate outputs.

B.2 Prompt Engineering Technologies

We uses the classical three-shot templates from each benchmark with the standard "Let's think step by step." reasoning instruction for CoT, and Direct prompting follows the official LiveCodeBench template for competitive programming tasks, whereas simpler problems are issued with their original prompts.

B.3 Agent-based Generation Approaches

To ensure a fair comparison across different agent-style methods, we unify the total number of iterative interactions to 9 rounds. Specifically, the debugging iterations are set to $t = 3$ with $k = 3$ for overall rounds in MAPCODER. PAIRCODER uses 3 clustered plans and $c = 3$ repair attempts for Driver. As for LPW, we follow its two-stage design: the solution-generation stage performs 9 verification iterations, and the code-implementation stage executes 9 correction iterations.

B.4 Repair-based Approaches

In SELF-REPAIR, we sample three solutions for each problem, execute them on the public tests, and repair those that failed. Specifically, each failed program is explained once, and three repair attempts are generated. For LDB, we set the repair iteration budget for incorrect solutions to 9.

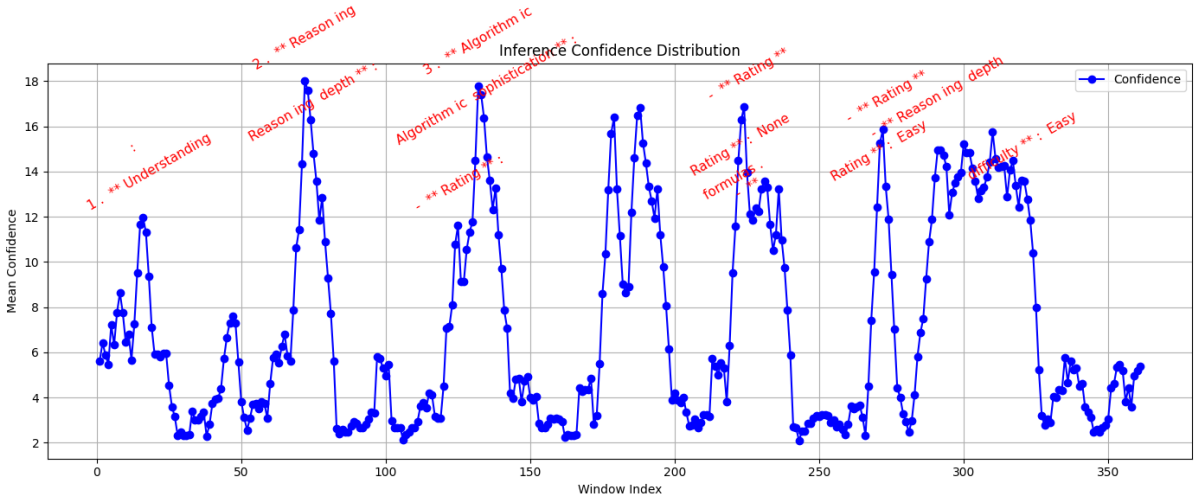

Figure 5: Confidence across windows with template tokens.

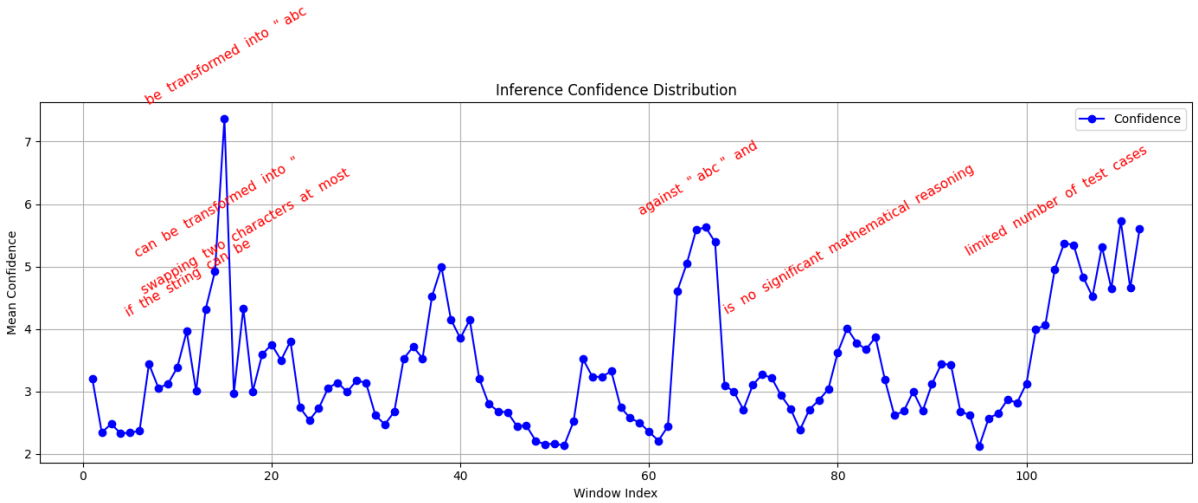

Figure 6: Confidence across windows without template tokens.

B.5 Tree-based Exploration Approaches

For TOT, we follow the original paper's passage-generation scheme: the model first produces five zero-shot CoT solution plans, from which we choose the highest-scoring plan to generate five candidate solution codes, and then select the code with the highest score. For CODETREE, we adopt the exploration strategy guided by the best-reported critic agent, as described in the original work, and set the generation budget to 9.

B.6 Confidence-based Approaches

Since the original HONESTCODER implementation does not include experiments on closed-source LLMs such as GPT-series models, we re-implement the method following its code repository. As final hidden-states are inaccessible for closed-source LLMs, we approximate semantic similarity by encoding the codes using the `text-embedding-3-large` model, which may introduces deviations. Following the best weights reported for GPT-4o in the repository, we set the four modality coefficients $(\alpha, \beta, \gamma, \delta)$ to $(0.85, 0.1, 0, 0.05)$ with the sampling count to $N = 9$, and evaluate the pass@1 over the top-5 highest-scoring generated programs. For SELF-ASK methods, we sample 9 candidate solutions for each query and ask the model to assign a confidence score from 1 to 5, where higher scores indicate greater likelihood of passing all tests. For each problem, we rank all generated programs by their scores and select the top five candidates for final evaluation; We uniformly sample five programs when more than five share the highest score.

C Confidence Analysis

In this section, we analyze the relevance between model-assigned confidence score and true task difficulty on LCB. We use LCB because it provides official verified difficulty labels—easy, medium, and hard—which align exactly with our own categorization scheme. Moreover, its difficulty distribution is relatively balanced (142 easy, 168 medium, 90 hard tasks), making it suitable for difficulty-sensitive analysis. For all subsequent experiments involving difficulty comparisons, we sample **90** problems from each difficulty level using a fixed random seed **42**. We further distinguish template-token confidence from inference-token confidence and examine how reasoning behavior differs considering template tokens.

C.1 Confidence and Difficulty Correlation

We analyze the relationship between a model's average inference confidence and the ground-truth difficulty labels in LiveCodeBench for GPT-4o-mini and GPT-4.1-nano. The two models exhibit notably different trends. GPT-4o-mini demonstrates a clear tendency where the central confidence values decrease as the difficulty increases. Statistical analysis confirms a moderate negative correlation between confidence and difficulty with Pearson Coefficient $r = -0.476$, $p = 0.0000$ and Spearman Coefficient $\rho = -0.471$, $p = 0.0000$, indicating that the model becomes less confident when facing harder problems. In contrast, GPT-4.1-nano shows no obvious visual trend, and the measured correlations are weak with Pearson Coefficient $r = -0.077$, $p = 0.2044$ and Spearman Coefficient $\rho = -0.172$, $p = 0.0045$. This suggests that its confidence is less sensitive to problem difficulty. The differences observed across models also suggest that their underlying knowledge and reasoning capabilities lead them to perceive task difficulty in notably different ways.

C.2 Impact of Template Token Removal

The average token confidence per window using GPT-4o-mini on CodeContest task 1575-A is illustrated in Fig 5 and Fig 6, before and after excluding template tokens. When all tokens are considered, high-confidence regions are dominated by fixed content, such as the required analysis angles specified in the prompt. These template tokens exhibit high confidence because the model consistently follows the prompt's instructions, often producing bullet-point style outputs with minimal actual reasoning. After removing template tokens, high-confidence windows correspond to more meaningful analysis, highlighting genuine understanding of the problem—for instance, recognizing that the task requires minimal mathematical inference. This suggests that focusing on inference tokens provides a more faithful reflection of the true problem difficulty.

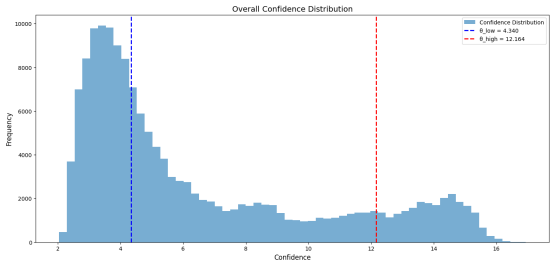

Figure 7: Overall Inference Confidence Distribution with template tokens.

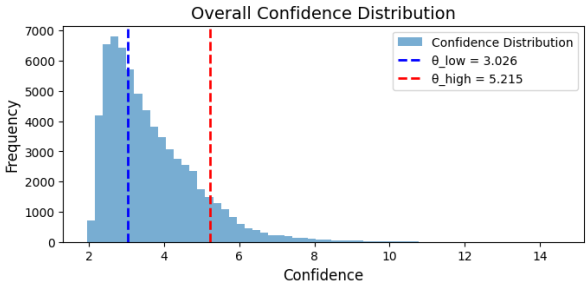

Figure 8: Overall Inference Confidence Distribution without template tokens.

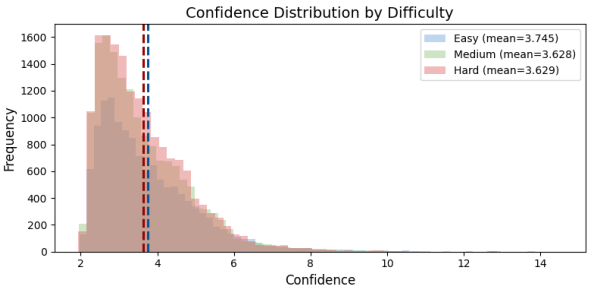

Figure 9: Inference Confidence Distribution by Difficulty.

C.3 Confidence Distribution

We analyze the confidence distribution using GPT-4o-mini. Fig 7 and Fig 8 shows the overall confidence histogram with and without template tokens respectively, where the blue and red dashed lines denote the two cluster centers obtained via K-means. The distribution reveals that most tokens

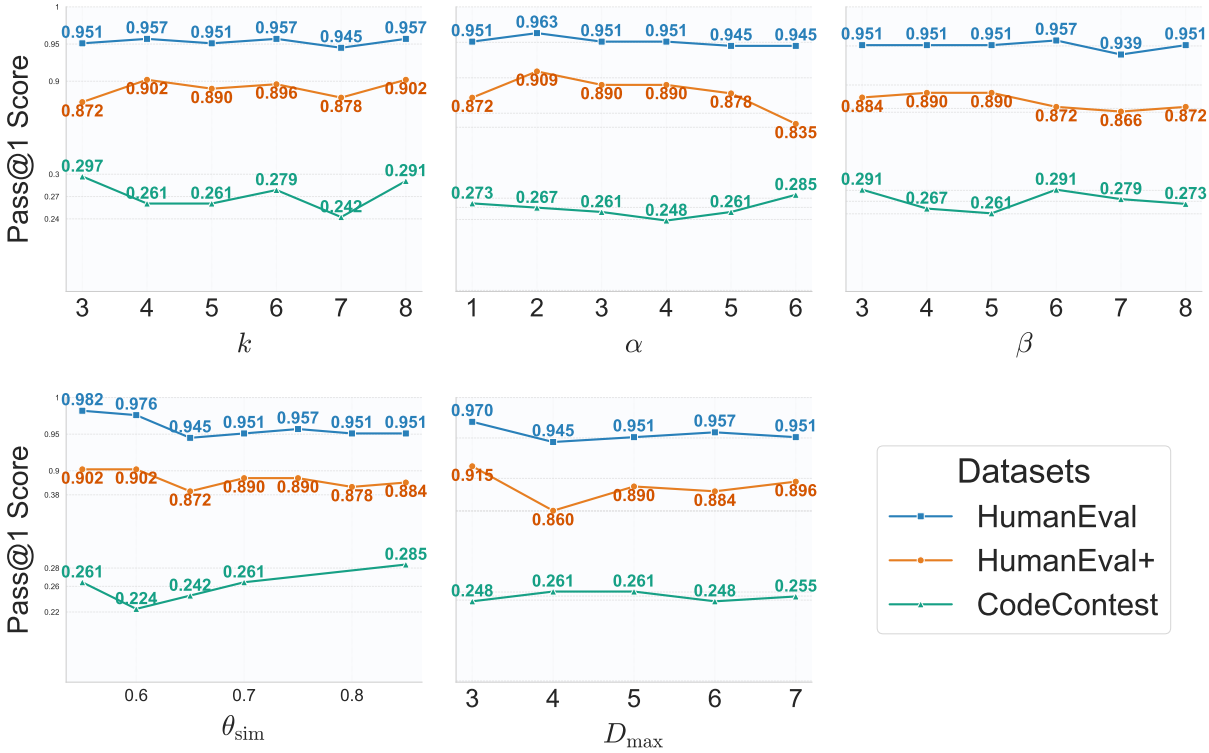

Figure 10: Impact of Hyperparameters on GPT-4.1-nano Performance.

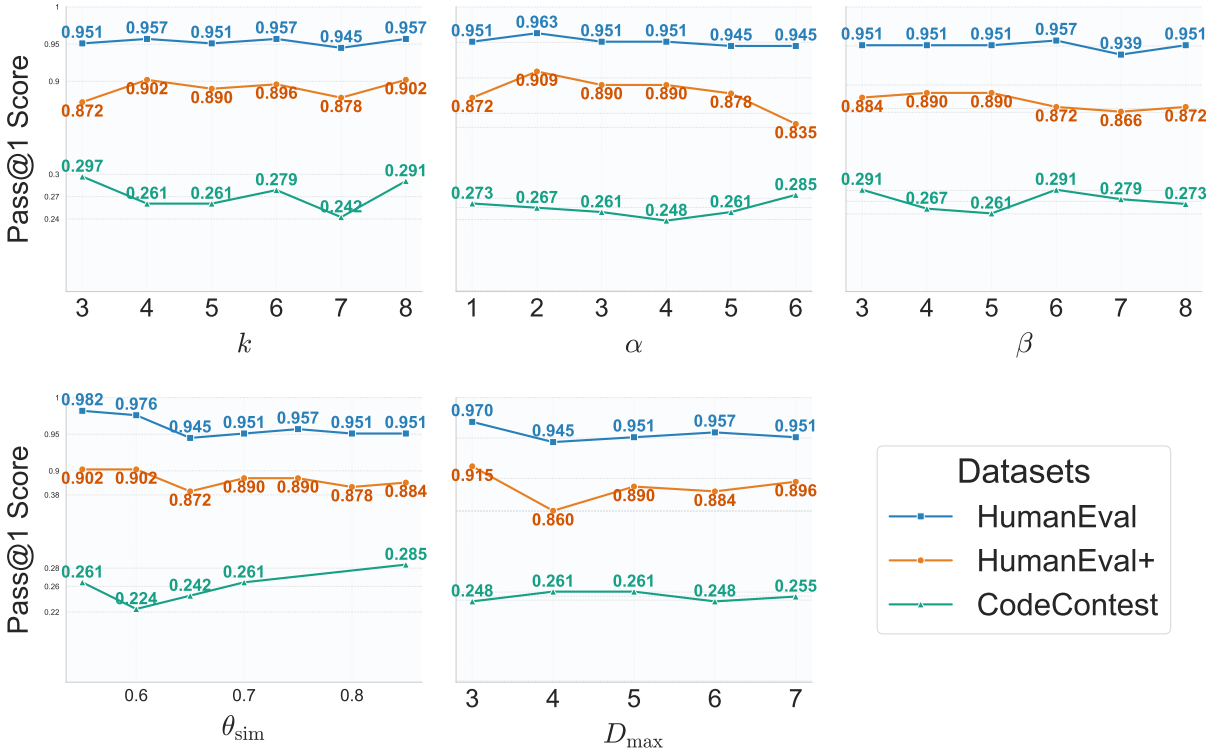

Figure 11: Impact of Hyperparameters on GPT-4o-mini Performance.

fall into the low or medium-confidence region, suggesting that the model is often uncertain during generation.

Compared with Fig 7, the proportion of high-confidence tokens drops substantially. This further confirms that the model, after instruction fine-tuning process, tends to produce aesthetically structured outputs and becomes overly certain about

fixed template tokens—often without any genuine reasoning involved. Eliminating these tokens is therefore justified, as it prevents such superficial certainty from distorting the analysis. Fig 9 further breaks down confidence distributions by ground-truth difficulty levels. For easy problems, low-confidence tokens appear less frequently, whereas hard problems exhibit a noticeably larger portion of low-confidence tokens. This pattern suggests that model confidence systematically decreases as task difficulty increases.

D Hyperparameter Sensitivity Analysis

In this section, we investigate the impact of other hyperparameters in MavenCoder on overall performance. Specifically, we conduct extensive experiments, analyzing the following parameters: (1) the top k value of log-probabilities used for confidence estimation, (2) the Beta distribution parameters α and β for modeling outcome contribution, (3) the similarity threshold θ_{sim} in plan tree construction, and (4) the maximum depth $D_{\max}$ of the plan tree.

It is worth noting that the configurations used in our main experiments are selected based on empirical experience or prior work, and may not always be fully optimized. Nevertheless, experimental results still show that MavenCoder achieves strong performance, further demonstrating the robustness and effectiveness of our proposed method.

Fig. 11 and Fig. 10 respectively present the pass@1 performance of GPT-4o-mini and GPT-4.1-nano under various hyperparameter settings on HumanEval, HumanEval+, and CodeContests. We observe that, larger values of k (e.g., $k = 8$) generally yield better performance, as incorporating more top-ranked token probabilities produces a more stable and robust confidence estimate. The parameters α and β control the skewness of the Beta distribution; smaller α and larger β (e.g., $\alpha \in \{2, 3\}$, $\beta = 5$) perform better, supporting our intuition that early-stage reasoning steps are more critical than I/O-related steps. Meanwhile, θ_{sim} and $D_{\max}$ govern the branching factor and depth of the plan tree. Increasing them within a reasonable range consistently improves performance, indicating that deeper and more diverse decomposition helps address complex tasks, although gains eventually plateau due to the trade-off between exploration and efficiency.

E Detailed Analysis

In this section, we conduct a detailed analysis of multiple methods on competitive programming problems. We examine their pass/fail outcomes, including error types on failed problems, as well as algorithm categories and difficulty labels for successfully solved cases. This analysis provides a fine-grained view of model behavior on contest-style tasks and highlights how different problem characteristics and failure patterns correlate with performance.

CodeContest	**WA**	**TLE**	**RE**	**Total**
Qwen3-Coder+	83 (93.4%)	4 (4.5%)	2 (2.2%)	89
GPT-4o-mini	114 (92.7%)	3 (2.4%)	6 (4.9%)	123
GPT-4.1-nano	101 (91.0%)	6 (5.4%)	4 (3.6%)	111
LiveCodeBench	**WA**	**TLE**	**RE**	**Total**
Qwen3-Coder+	27 (48.2%)	25 (44.6%)	4 (7.1%)	56
GPT-4o-mini	111 (64.2%)	61 (35.3%)	1 (0.6%)	173
GPT-4.1-nano	81 (67.5%)	35 (29.2%)	4 (3.3%)	120

Table 7: Error Types and Proportions.

E.1 Failed Types

To better understand the failure behaviors of different models, we analyze the distribution of three error types: Wrong Answer (WA), Time Limit Exceeded (TLE), and Runtime Error (RE). As shown in Table 7, CodeContest is dominated by WA across all models, with more than 90% of the failures arising from incorrect or incomplete logic. This reflects that competitive-programming problems typically require multi-step reasoning and remain a key challenge for code generation.

In contrast, failed types on LiveCodeBench introduces additional TLE failures beyond WA, indicating that while the model is able to produce functionally correct solutions, it often fails to meet stricter time constraints, suggesting that more efficient algorithmic strategies remain unexplored.

Runtime errors remain relatively rare across all settings, suggesting that modern models can reliably adhere to input–output specifications and generally produce syntactically valid, executable code. Overall, future efforts should focus on achieving functional correctness while simultaneously ensuring high efficiency to generate high quality codes.

E.2 Performance on varying Difficulties

Fig 12 to Fig 14 present the distributions across difficulty levels. For each model, the left panel shows the performance achieved by different methods on LiveCodeBench, while the right panel displays the result of MavenCoder on CodeContest problems with varying Codeforces (CF) ratings.

Across all models, MavenCoder consistently

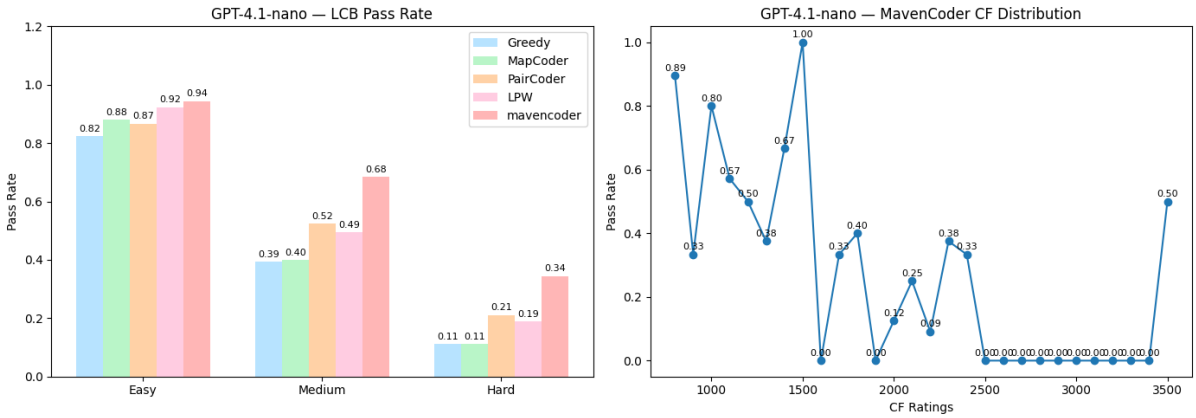

Figure 12: Pass@1 Accuracy Distribution by Difficulty for GPT-4.1-nano.

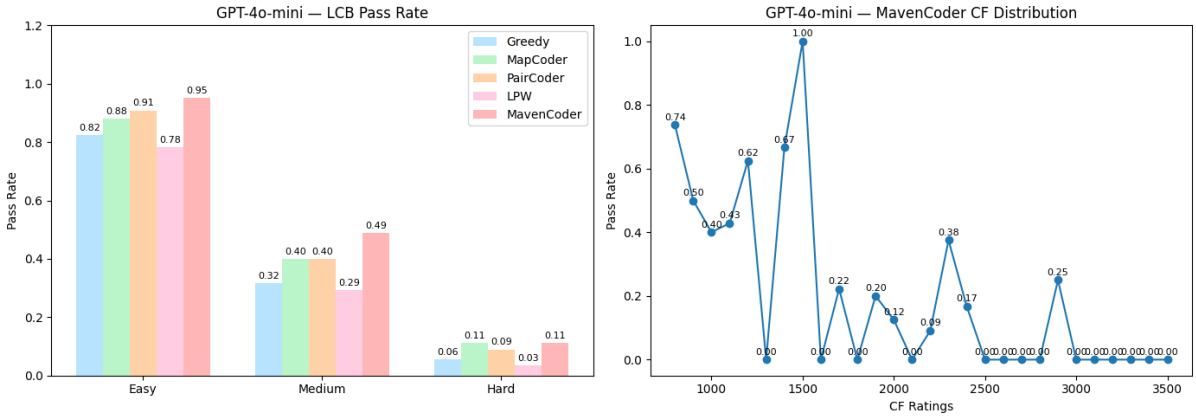

Figure 13: Pass@1 Accuracy Distribution by Difficulty for GPT-4o-mini.

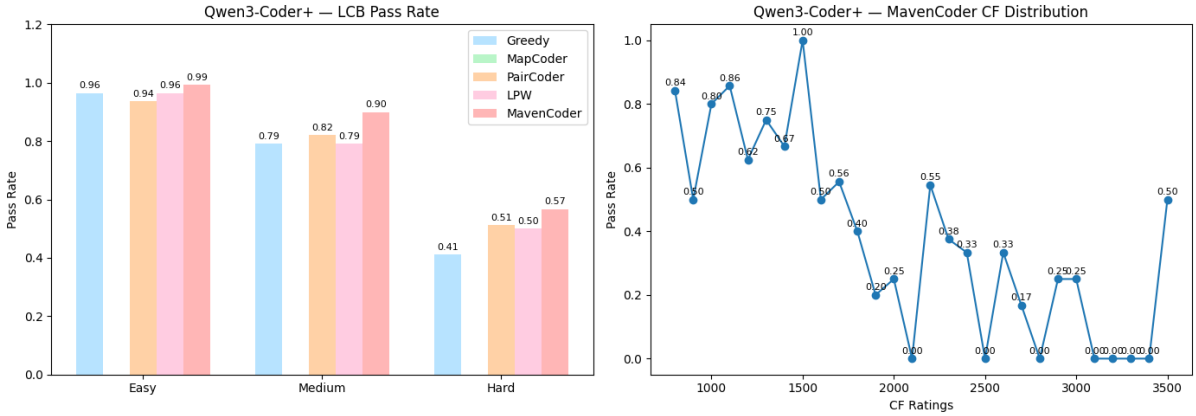

Figure 14: Pass@1 Accuracy Distribution by Difficulty for Qwen3-Coder+.

achieves higher results than other approaches, demonstrating that adaptive reasoning enhances solution quality and assists models better match the problem's structure. The right-hand plots further reveal that most successfully solved tasks fall below the 2000–2300 CF rating range. According to Codeforces official rating tiers[6], this corresponds to: **Purple (Candidate Master, 1900 $\leq$ rating<2100)** or **Orange (Master, 2100 $\leq$ rating<2300)**. These levels represent contestants capable of achieving strong results in provincial even national programming competitions, indicating that

[6] https://codeforces.com/ratings

MavenCoder enables the model to reach problem-solving abilities comparable to highly skilled human programmers. Though it has been reported that OpenAI's o3 model has already won a gold medal in the 2024 IOI (El-Kishky et al., 2025), we believe that MavenCoder, with advanced backbone LLMs like o3, can achieve even higher results in such international contests.

Algorithm 1: PLAN TREE CONSTRUCTION

Input: Problem context c, large language model LLM
Output: Plan Tree T

$P_{sub} \leftarrow \text{SubProblems}(\text{LLM}(c))$;
for $P_i \in P_{sub}$ **do**
  $e_i \leftarrow \text{Embedding}(P_i)$;
  $\mathcal{D} \leftarrow (P_i, e_i, s_i)$;
Initialize queue $Q \leftarrow P_{sub}$;
`// - Decomposition -`
while Q *not empty* **do**
  $P_{cur} \leftarrow Q.\text{pop}()$;
  if $\mathit{depth}(P_{cur}) \geq D_{max}$ **then**
    continue;
  $P_{sub} \leftarrow \text{SubProblems}(\text{LLM}(P_{cur}))$;
  if $P_{sub} = \varnothing$ **then**
    $\text{Solve}(P_{cur})$
    continue;
  `// Check for new sub-problems`
  for $P_j \in P_{sub}$ **do**
    $\mathbf{e}_j \leftarrow \text{Embedding}(P_j)$;
    $sim_{max} \leftarrow \max_{\mathbf{e_i} \in D} \frac{e_j \cdot e_i}{\|e_j\| \|\mathbf{e_i}\|}$;
    if $sim_{max} < \theta_{sim}$ **then**
      $\mathcal{D} \leftarrow (P_j, e_j, s_j)$;;
      $Q.\text{push}(p)$;
return *linearized(T)*;

E.3 Performance on Different Algorithm Tags

Fig 16 summarizes the performance of three models across algorithmic categories on CodeContest. All models perform well on basic categories such as strings, two-pointers, where reasoning demands are relatively low. However, success rates drop notably for categories requiring more complex structures or mathematical reasoning—such as dynamic programming, combinatorics, and graph matchings—highlighting persistent challenges in advanced algorithmic reasoning. Moreover, Qwen3-Coder+, empowered by extensive code corpora, handles more algorithmic categories and achieves higher success rates, highlighting the importance of broader code-related knowledge.

Table 8: Pass@1 comparison on LCB-Hard and HumanEval+ with different plan strategy.

Method	LCB-Hard	HumanEval+
Direct (Greedy)	**11.1**	**82.9**
CoT	4.4	75.6
LPW	**11.1**	72.0
Self-Planning	4.4	81.1
Plan Tree	**18.9**	70.1

F Analysis of Plan Trees

F.1 Construction Procedure

Detailed construction of Plan Tree is provided in Algorithm 1.

F.2 Effectiveness of Plan Tree

We compare GPT-4.1-nano across different planning strategies on LCB-Hard problems and HE+. As shown in Table 8, Plan Tree achieves the highest performance on LCB-Hard, indicating **its strength in decomposing complex problems that require deeper reasoning**. However, the performance sharply drops on HE+, indicating that **excessive planning may introduce unnecessary overhead on simpler tasks**.

G Plan Quality Comparison

Figure 15 presents an illustrative comparison of plan quality between MavenCoder and prior methods. On the HumanEval/40 task, Qwen3-Coder+ with LPW or Self-Planning adopts a brute-force strategy with triple nested loops, leading to high computational overhead and relatively simplistic plans. In contrast, MavenCoder produces a more efficient plan based on a two-pointer optimization, demonstrating deeper problem understanding and more detailed reasoning.

H Generalization to Advanced Models

To evaluate the generalization ability of our framework, we further conduct experiments using a recent model, GLM-4.7 (Glm et al., 2024), as the backbone. Since GLM-4.7 is a reasoning-oriented model without direct access to internal states, we instead employ prompt-based difficulty classification and score estimation for the confidence-related components.

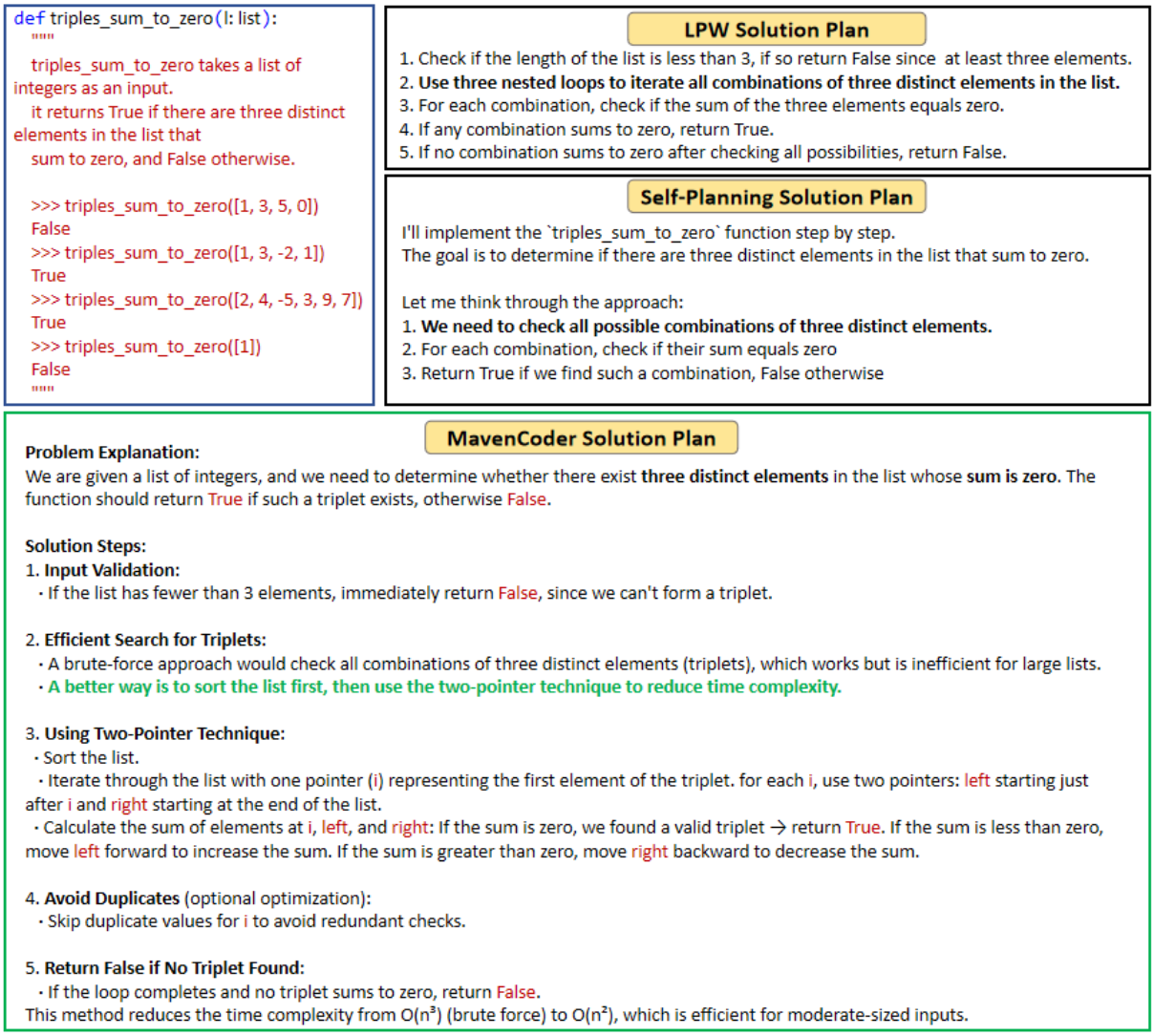

Figure 15: Comparison of Solution Plans using Qwen3-Coder+ on HumanEval/40. Three methods generate plans for the same tasks, while MavenCoder produces more detailed plans and employing more effective optimization strategies than the alternatives.

Method	HE	HE+	MBPP	MBPP+	LCB	CC
Direct (Random)	82.6	78.1	90.7	78.8	69.3	16.7
CoT	79.9	73.8	91.8	77.5	61.0	13.9
CodeTree	97.0	89.0	98.4	**82.5**	75.5	33.9
MavenCoder	**98.2**	**92.1**	**99.5**	**82.5**	**85.8**	**49.1**

Table 9: Performance comparison of different methods using GLM-4.7 as the backbone across multiple benchmarks.

From table 9, GLM-4.7 consistently achieves the best performance across all benchmarks when integrated with MavenCoder. This demonstrates that MavenCoder is largely model-agnostic and can effectively adapt to different newer backbone architectures. The performance gains further show that even for black-box LLMs, adaptive planning can be effectively achieved through direct classification and score-based evaluation, demonstrating the strong generalizability of our framework.

I Prompts

This section provides all prompts used in our experiments across different agents for MavenCoder.

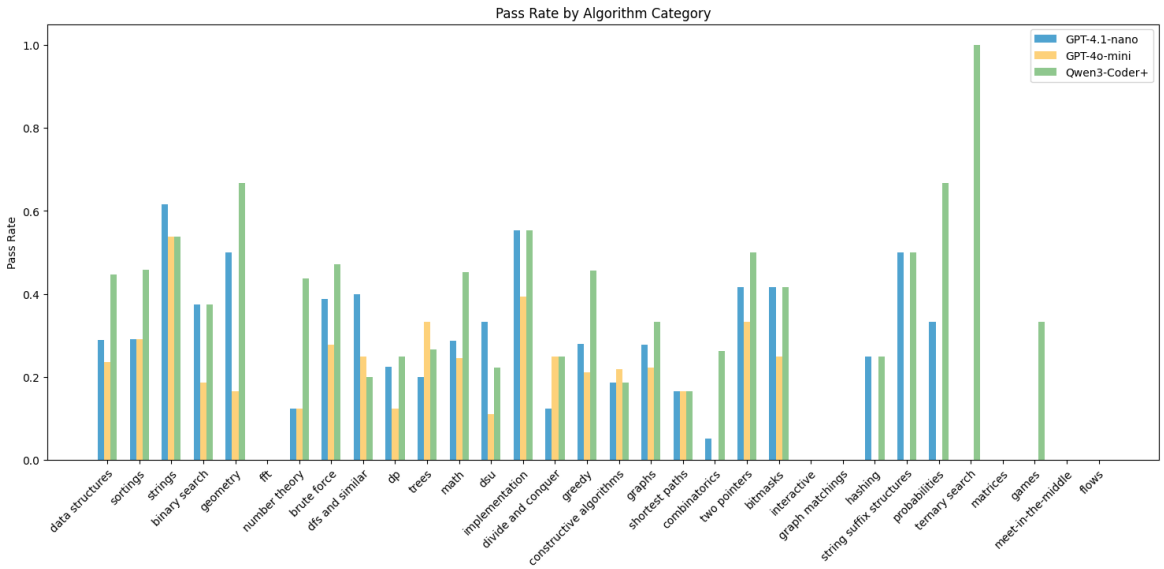

Figure 16: Distribution by Algorithm Tags on CodeContest.

You are an expert in evaluating problem difficulty for competitive programming problems.

Evaluation Dimensions
Evaluate the difficulty based on the following aspects:

1. Understanding complexity - How hard is it to correctly interpret the problem?
2. Reasoning depth - How many logical steps are required to solve it?
3. Algorithmic sophistication - What is the expected algorithmic or conceptual difficulty?
4. Mathematical difficulty - How much mathematical reasoning, formulas, or abstract modeling is required.
5. implementation_difficulty - How hard it is to implement, test, and debug the solution.

Problem:
{problem}

Figure 17: Model-Adaptive Difficulty Assessment Prompt

Given a programming problem, analyze the problem through these steps to provide a comprehensive analysis.

1. **Problem Comprehension**
   - Understand the intend of this problem
   - Identify core requirements.

2. **Example Explanation**:
For each provided example, explain how each sample input will yield the sample output through logical step-by-step inference.

3. **Edge Case Synthesis**:
   - Consider data constraints and inform how to properly handle edge cases

[Problem]:
{problem}

Now, provide your comprehensive problem analysis

Figure 18: Self-Reflection Prompt

You are given a programmming problem:
Problem:
{problem}

Please briefly give a concrete explanation for the problem, and provide logical steps without codes to solve the problem. Avoid overthinking. Sequence operations logically to solve the problem, start with **Solution Steps.**

Easy

You are given a programmming problem:
Problem:
{problem}

Your reflection for this problem:
{reflection}

Provide a solution approach with natural language without codes.

You should:
1. Understand the intend of problem and analyze key requirements from problem statement
2. Explain how public sample inputs yield the expected outputs.
3. Think of proper algorithm tags, map each requirement to relevant algorithm tags and identify necessary operations implied by the tags
4. Sequence operations logically to solve the problem.

Medium

Figure 19: Generate Solution Plan Prompt

You previously generated a plan consisting of multiple steps and then produced modular code based on that plan. After Executing the code with sample input, and below are the partial actual console outputs produced by each step.

Your task:
1. Identify and compare **each step's** output against the expected logic described in the original plan.
2. Determine whether each step is correct.
3. Use **0 for incorrect** and **1 for correct**.
The result of the code execution may only contain the output of **some steps (not all)**. You need to evaluate **each step** of the solution plan.

Return the evaluation strictly in the following **YAML format** (No extra text outside):

```yaml
 - step k:
   name: |
     Description of the k step
   score: 1
   reason: |
     Brief explanation why step k is correct or incorrect
```

Attention: Ensure that **the k of the last step** is equal to **the total number** of solution steps.

Figure 20: Fact Evaluation Prompt

You are an expert Python programmer. Your shoul generate Python code to solve the given problem based on the given **solution steps**. Follow these strict requirements:

1. **Modular design**: Each step in the solution plan must be implemented as a separate function.
2. **Step execution**: After defining all step functions, you must use **the function provided in the question** that calls each step function **in order**, following the logic of the provided solution steps.
3. **Input/output visibility**: Every function must **print its input and output values** clearly for debugging purposes.
4. **Self-contained code**: The final code should be executable without any external dependencies beyond standard Python libraries.

Task: Write Python code that implements these steps exactly as described. Ensure the code is modular, prints inputs and outputs for each step, and calls the functions sequentially at the end.

<one-shot Example>

Figure 21: Modular Code Generation Prompt

Given a programming problem, you have generated a solution approach for the problem. However, you reviewd this approach and found some mistakes. Please generated a new correct natural-language approach without codes.

You should avoid to repeat the same logic errors in the original approach.

[Problem]:
{problem}

[Original Approach]:
{solution plan}

[Your Review Explanation]:
{review}

Figure 22: Rebuild Solution Plan Prompt

Question:
{problem}

Approach:
{solution plan}

Format:
Read the inputs from stdin solve the problem and write the answer to stdout (do not directly test on the sample inputs). Enclose your code within delimiters as follows. Ensure that when the python program runs, it reads the inputs, runs the algorithm and writes output to STDOUT.

```python
# YOUR CODE HERE
```

Answer: (use the provided format with backticks)

Figure 23: Code Implementation Prompt

Task: You are given a programming problem and a possible implement approach. Please analyze programming errors and provide natural-language repair guidance for the last buggy code.

[Problem]:
{problem}

[Approach]:
{approach}

[Submission History]:
{execution_results}

First, ensure the code correctly accepts input and produces the proper output format, for instance, if the function call is missing, the actual output of given code will be empty, and you should suggest to call it explicitly.

Second, you should clearly identify the **root cause** of failure in the implementations, specify **exact location(s)** of critical flaws and explain **why** these cause failure with concrete evidence from: problem constraints, historical commit context and failure scenarios.
Then, provide **sequential repair instructions** that specify content to fix and ensure functional correctness.

Figure 24: Generate Fix Instruct Prompt

You are an AI planning and reasoning assistant. You will receive a **hierarchical plan tree** designed to decompose a programming problem. The tree is formatted as follows:

Structure Definition
· The root starts with:
To solve this problem, we can further break it down into more sub-problems:
· Each sub-problem node is prefixed with:
- sub-problem X:
where X uses a hierarchical index (e.g. 1, 1-2, 2-3-1…), indicating its dependency relations.
· Each sub-problem node may include:
A **Current Problem** description (the task for this node to solve)
A **Solution** section (the proposed approach for **current** sub-problem)
Or further decomposed child sub-problems

Semantics of the Plan Tree

· Higher-level sub-problems represent broader tasks.
· Deeper indexed nodes are **supporting sub-problems**, which must be solved first because they enable the parent.
· The goal is to **integrate** all the sub-solutions from leaf nodes upward, forming a complete and logically coherent solution to the root problem.

Your Task
1. Read and understand the entire plan tree.
2. Ignore the hierarchical numbering in the final write-up (it is only for dependency guidance).
3. Merge all **leaf-node Solution sections** properly into a single, holistic **final solution plan**.
4. Ensure the final solution: Preserves the logical order implied by the hierarchy, Removes redundancy and repeated questions, Enhances clarity, structure, and reasoning consistency, Writes in clean and professional technical language

Figure 25: Tree-Structured Reasoning Prompt

You are given a problem:
{problem}

A Buggy Implementation for the problem:
{error_code}

The Approach that buggy codes implements:
{solution plan}

Concrete Fix Instructions:
{explanation}

Your Task is to generate a corrected, runnable program that EXACTLY implements the provided fix instructions, ensure the solution addresses all issues mentioned in the fix instructions and pass all hidden test cases (**Your code should not directly test on the sample inputs**). Output ONLY the corrected Python code without any explanations, comments, or additional text.

Figure 26: Bug Fix Prompt

You are a programming helper. Your task is to write a **Solution Plan** for the problem by divide and conquer in multiple turns.

Rules:
Given a programming problem to solve, and current problem, you need to focus on solving the current problem in each turn and attempt to decompose it into sub-problems if it is complex.
For Each turn:
· If the current problem is simple, directly solve it in the current turn.
· If the current problem is complex, divide it into multiple sub-problems.
Do not include any codes or implementation details in your response.

For Each Turn, Your response must contain:
1. Current Problem: what is the current problem to solve.
2. · If the current problem is complex to solve, list divided sub-problems.
· If the current problem is simple and direct, provide a concrete solution to solve it.
If a problem:
· No further reasoning required — the problem can be solved directly without multi-step logic or intermediate derivations.
· Operates on local or concrete data — it deals with specific values, strings, or conditions rather than abstract concepts.
· Independently solvable — it can be completed without relying on other sub-tasks or broader context.
Then, it is considered a **simple problem**.

<one-shot Example>

Figure 27: Problem Decomposition Prompt

You are a programming helper. Your task is to write a **Solution Plan** for the given problem.

Rules:
· You are solving the **current problem**, not the full task.
· Use provided sub-problems and their solutions as references if available.
· Do NOT output code. Only reasoning and planning steps.
· If all sub-problems are already solved, integrate their reasoning to form a coherent overall solution.

Problem to Solve:
{problem}

Current Problem:
{node.cur_problem}

{reference_text}
Now, using the information above, provide the final reasoning and integrated solution.

Figure 28: Resolve Leaves Prompt

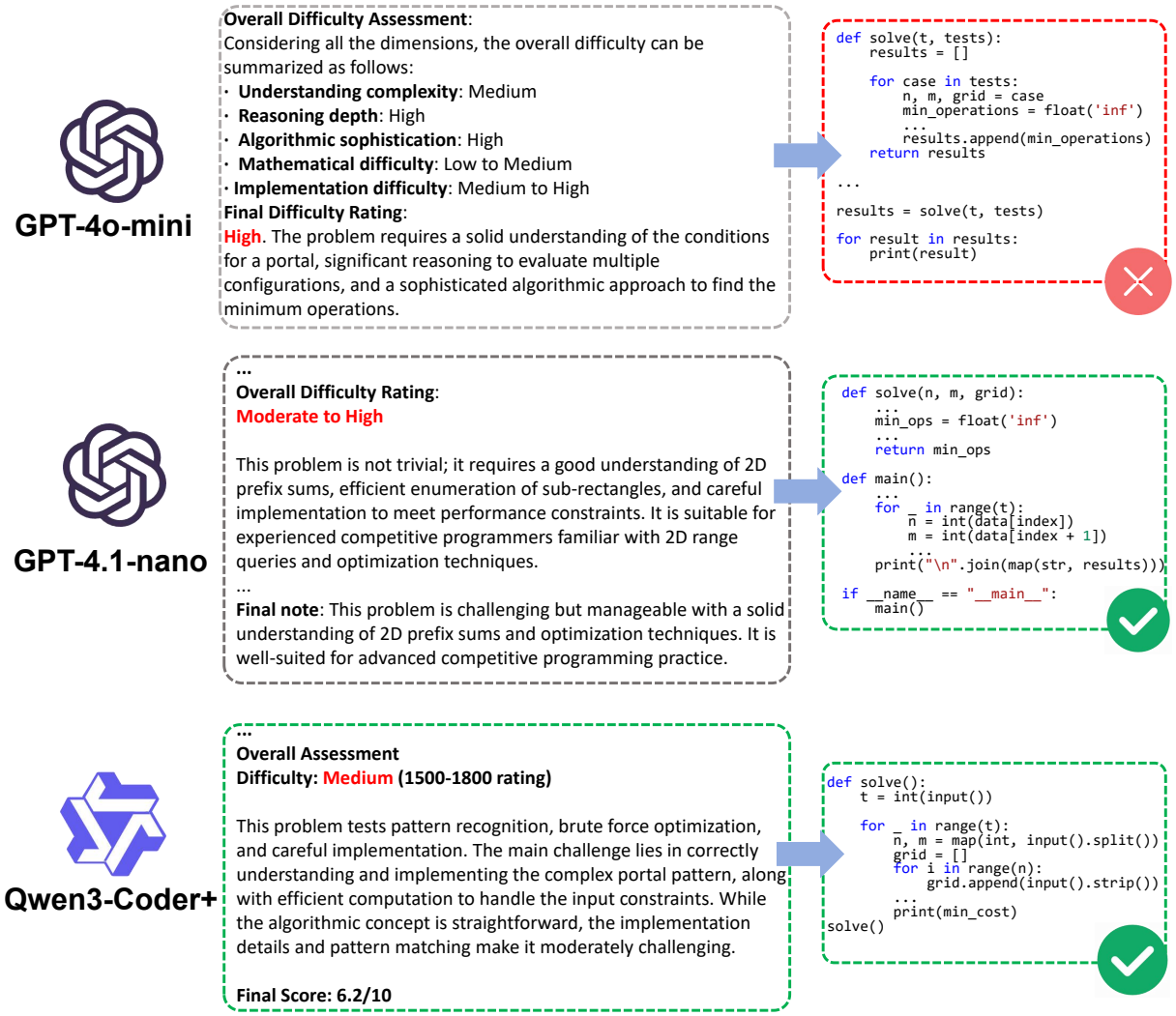

Figure 29: Difficulty assessments on Code Contests 1579 C (Ticks).The models categorize the task as hard, medium–hard, and medium, respectively, leading to divergent code outcomes. This illustrates that models rely on their internal knowledge to judge task difficulty, underscoring the necessity of model-adaptive planning.