SACP-CODER: STATE-BASED ADAPTIVE COMPUTATION FOR PROGRESSIVE CODE GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Code generation performance has been considerably improved by recent agentic systems, but such gains are often accompanied by increased token consumption. To better adjust computation, existing adaptive methods either select a solving method before generation or repeatedly revise candidate code using test feedback under a predefined strategy. However, these methods tend to use the same solving strategy as a problem's solving state evolves, overlooking changes in computational needs and potentially wasting tokens on unnecessary computation. To better balance performance and cost, we propose SACP-Coder, a state-adaptive framework that dynamically adjusts computation to match evolving solving needs. We design state-adaptive mechanisms, including retrieval augmentation, problem reformulation, and dual-level correction, and progressively adapt their use to the solver state characterized by candidate solutions and public-test execution outcomes. In particular, problem reformulation facilitates solving challenging problems by deriving a simpler problem with the same core algorithmic structure and transferring its solution pattern back to the original problem. Experiments on four code generation benchmarks across three LLMs demonstrate a favorable performance–cost trade-off. SACP-Coder reduces average token consumption by over **76**% relative to comparably accurate baselines on each LLM backbone.

1 INTRODUCTION

Code generation, which aims to synthesize functionally correct programs from natural-language descriptions (Dong et al., 2024; Jiang et al., 2026), has benefited from advances in multi-agent systems that improve performance through collaboration among specialized agents, such as planners, coders, verifiers, and debuggers (Islam et al., 2024; Xu et al., 2026). However, the amount of reasoning and computation required for effective code generation can vary considerably across different instances. Applying the same complex method to all problems may therefore incur unnecessary token costs and may even hurt performance on some simple problems (Wu et al., 2026; Li et al., 2026).

To reduce unnecessary computation, existing studies have explored how to adjust computation across different code generation instances, which broadly fall into two groups: pre-generation routing and fixed-strategy iteration, as illustrated in Fig. 1(a) and (b). Pre-generation routing methods (Li et al., 2026) select a solving strategy before code generation based on problem characteristics or estimated difficulty, but cannot revise an inappropriate initial choice. Fixed-strategy iteration methods (Chen et al., 2024b; Zhu et al., 2025) first generate candidate code and then repeatedly revise it using test feedback under a predefined strategy, which remains fixed throughout solving a given problem. Such fixed-strategy refinement can lead to repeated computation that does not effectively address the current failure state.

These limitations highlight a key insight: efficient code generation requires adapting the solving strategy to the evolving solver state as solving progresses on a given problem. Accordingly, we ask: ***How can computation be dynamically adjusted according to the evolving solver state to improve inference-time efficiency?***

To address this question, we propose SACP-Coder, a **S**tate-based **A**daptive **C**omputation for **P**rogressive coding framework that dynamically adjusts inference-time computation according to the evolving solving state. As illustrated in Fig. 1(c), SACP-Coder formulates code generation as a sequential process over solver states and progressive computation actions, where the solver state

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

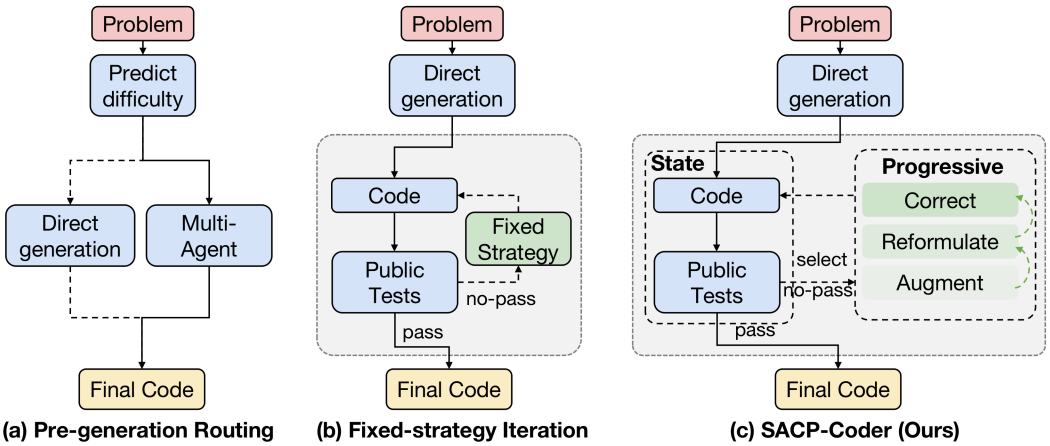


Figure 1: Computation adjustment methods for code generation. (a) Pre-generation routing selects a strategy before generation. (b) Fixed-strategy iteration revises code under a predefined strategy. (c) SACP-Coder progressively adapts computation actions to the evolving solver state.

is continuously updated using the current candidate solution and its execution outcomes on public tests. Starting from a low-cost direct solution, SACP-Coder maps the observed solver state to an appropriate computation action and progressively updates the action as the solving state changes. Specifically, SACP-Coder provides state-adaptive solving mechanisms including retrieval augmentation, problem reformulation, and dual-level correction, allowing additional computation to better match the current solving needs. For challenging problems, reformulation provides a stronger solving mechanism by deriving a simpler problem with the same core algorithmic structure and transferring its solution pattern back to the original problem.

We evaluate the proposed method on four code generation benchmarks against representative single-agent reasoning, multi-agent collaboration, and adaptive computation methods. Extensive experiments across different large language models demonstrate that SACP-Coder achieves a favorable performance-cost trade-off and maintains strong solving capability on challenging coding problems. Our main contributions are summarized as follows:

- Guided by the insight that aligning computation with the evolving solver state is essential for efficient inference-time computation, we propose **SACP-Coder**, a State-based Adaptive Computation for Progressive Coding framework that dynamically adapts computation throughout the solving process.
- We design state-adaptive solving mechanisms, including retrieval augmentation, problem reformulation, and dual-level correction, to support progressive solving in SACP-Coder. Among them, reformulation provides a way to solve challenging problems by transferring solution patterns from simpler, structurally homologous problems.
- Extensive experiments on four code generation benchmarks across three LLM backbones demonstrate a favorable performance-cost trade-off: SACP-Coder achieves equivalent or higher accuracy while reducing average token consumption by over **76**% relative to comparably accurate baselines on each backbone.

2 Related Work

Agentic Code Generation. Beyond direct prompting, agentic approaches improve code generation through iterative refinement and multi-agent collaboration. Self-Debugging (Chen et al., 2024b) uses code explanation and execution feedback to iteratively improve generated programs. Multi-agent systems extend these capabilities through specialized agent collaboration. MapCoder (Islam et al., 2024) coordinates example recall, planning, coding, and debugging, while MavenCoder (Xu et al., 2026) combines model-adaptive planning with multi-perspective verification. CodeTree (Li et al., 2025c) further organizes strategy exploration and solution refinement within a feedback-guided search tree. Beyond agent coordination, S* (Li et al., 2025a) combines parallel and sequential

test-time scaling with execution-grounded candidate selection. To adjust computation for different code generation problems, RoutingGen (Li et al., 2026) selects a solving approach before generation based on estimated difficulty, whereas AdaCoder (Zhu et al., 2025) and PaT (Yoon et al., 2026) begin with direct attempts and introduce additional planning or debugging when needed. Rather than relying on pre-generation strategy selection or repeated code revision, SACP-Coder adapts computation to evolving solver states through progressive transitions among different solving mechanisms.

Reasoning-Enhanced Code Generation. To address algorithmic reasoning challenges in code generation, existing studies have explored various enhancement strategies. Example-based approaches provide reusable solution patterns: retrieve-and-edit methods (Hashimoto et al., 2018) adapt retrieved examples to new inputs, while analogical prompting (Yasunaga et al., 2024) generates relevant problems, solutions, and knowledge to guide target problem solving. MapCoder (Islam et al., 2024) similarly incorporates recalled examples into its planning process. Beyond whole-problem examples, CodeChain (Le et al., 2024) promotes modular code generation and iteratively revises solutions using representative sub-modules from previous generations. Other work improves the organization of problem descriptions: StoryCoder (Jang et al., 2026) presents problem conditions as coherent narratives to support structured reasoning. In comparison, our reformulation mechanism derives a simpler, structurally related problem and guides the transfer of its solution pattern back to the original problem, facilitating algorithmic reasoning on challenging coding tasks.

Adaptive Test-Time Compute. Adaptive test-time computation aims to achieve a better balance between performance and computational cost by varying computational effort across inputs and during generation. Snell et al. (2025) shows that the effectiveness of test-time scaling strategies depends on problem difficulty, motivating input-dependent strategy selection. Damani et al. (2025) predicts the marginal benefits of additional computation to allocate inference budgets. Beyond input-based allocation, Aggarwal et al. (2023) dynamically adjusts sample counts using agreement among generated answers. At the model level, FrugalGPT (Chen et al., 2024a) uses model cascades to balance response quality and inference cost. Motivated by the benefits of adaptive computation demonstrated by these studies, we investigate the balance between performance and computational cost in code generation by considering solver state and evolving solving needs of different problems.

3 SACP-CODER: STATE-BASED ADAPTIVE COMPUTATION FOR PROGRESSIVE CODING

SACP-Coder progressively adapts computation actions to the evolving solver state. In this section, we first define solver states and computation actions in § 3.1, then present the progressive state-adaptive reasoning workflow in § 3.2 and the state-adaptive solving mechanisms in § 3.3. Fig. 2 and Fig. 3 illustrate the overall workflow and the individual solving mechanisms, respectively.

3.1 PRELIMINARY

We formulate program solving as a sequential decision process over solver states and computation actions. Let s denote a programming problem, $\mathcal{T}_s = \{(x^i, y^i)\}_{i=1}^{n_s}$ its public sample tests with n_s samples, and let M denote the LLM used by the solver agent. At step t, $f_s^{(t)}$ denotes the current candidate solution for s. For problems with available public sample tests, we introduce a **solver state** that explicitly characterizes the current solving progress from observable execution evidence. Specifically, we quantify the sample-level execution success of $f_s^{(t)}$ by the pass ratio

$$r_s^{(t)} = \frac{1}{n_s} \sum_{i=1}^{n_s} \mathbb{I}\Big[\mathrm{Exec}(f_s^{(t)}, x^i) = y^i\Big]. \tag{1}$$

Here, $r_s^{(t)}$ reflects the observed solving progress on the available samples. We further collect execution feedbacks $\phi_s^{(t)}$, including failing inputs, expected and observed outputs, and compiler or runtime errors when applicable. The solver state is then represented as

$$z_s^{(t)} = \big(f_s^{(t)}, r_s^{(t)}, \phi_s^{(t)}\big), \tag{2}$$

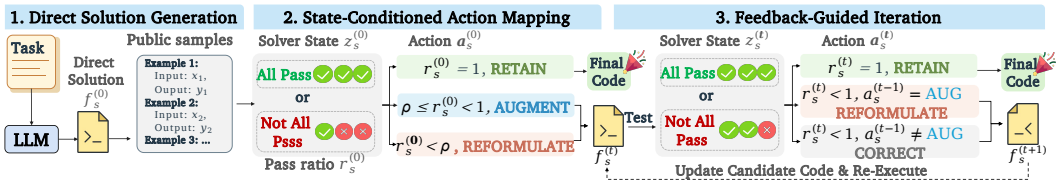


Figure 2: Progressive state-adaptive reasoning workflow of SACP-Coder. Direct solution generation is followed by state-conditioned action mapping and feedback-guided iteration using execution outcomes on public sample tests.

where $f_s^{(t)}$ specifies the current candidate solution, while $r_s^{(t)}$ and $\phi_s^{(t)}$ provide coarse-grained and fine-grained execution evidence, respectively. Given the current solver state $z_s^{(t)}$, SACP-Coder maps it to a computation action $a_s^{(t)} \in \mathcal{A}$, where $\mathcal{A} = \{\textsc{Retain}, \textsc{Augment}, \textsc{Reformulate}, \textsc{Correct}\}$ denotes the computation action space. For a non-terminal action, applying $a_s^{(t)}$ updates the candidate solution and consequently the solver state:

$$z_s^{(t)} \xrightarrow{a_s^{(t)}} z_s^{(t+1)}. \tag{3}$$

3.2 Progressive State-Adaptive Reasoning Workflow

Direct Solution Generation. To avoid unnecessary computation on problems that can be solved directly, SACP-Coder first attempts a direct solution from the original problem statement. Given a problem s, the solver agent uses its underlying LLM M to generate an initial candidate solution:

$$f_s^{(0)} = M(s). \tag{4}$$

Programming problems typically provide public sample tests. If no public sample tests are available, SACP-Coder directly returns $f_s^{(0)}$ without further computation. Otherwise, the generated solution is evaluated against the public test cases $\mathcal{T}_s$, yielding the initial pass ratio $r_s^{(0)}$ and execution feedback $\phi_s^{(0)}$. These observations are then used to construct the initial solver state

$$z_s^{(0)} = \left(f_s^{(0)}, r_s^{(0)}, \phi_s^{(0)}\right). \tag{5}$$

This direct attempt allows readily solvable problems to terminate after a single model call, while providing the initial solver state for further intervention when needed.

State-Conditioned Action Mapping. To tailor subsequent computation to varying solving progress across problems, we design a state-conditioned action mapping rule that assigns an appropriate computation action according to the observed Direct pass ratio $r_s^{(0)}$. Rather than manually setting the decision threshold, we calibrate ρ using validation problems that are not fully solved by direct generation:

$$\rho = \text{Median}\left(\left\{r_s^{(0)} \mid s \in \mathcal{D}_{\text{val}},\ 0 < r_s^{(0)} < 1\right\}\right). \tag{6}$$

Given the initial solver state $z_s^{(0)}$, SACP-Coder maps it to an initial computation action according to the rule π_ρ^{init}:

$$a_s^{(0)} = \pi_\rho^{\text{init}}\left(z_s^{(0)}\right) = \begin{cases} \textsc{Retain}, & r_s^{(0)} = 1, \\ \textsc{Augment}, & \rho \le r_s^{(0)} < 1, \\ \textsc{Reformulate}, & r_s^{(0)} < \rho. \end{cases} \tag{7}$$

This mapping adapts subsequent computation to the sample-level performance of the direct solution. A solution that passes all available public sample tests is retained without further model calls. For solutions with $\rho \le r_s^{(0)} < 1$, the observed partial success suggests that the current solution has captured useful problem structure, and SACP-Coder therefore applies $\textsc{Augment}$ to provide additional references without altering the original problem formulation. For solutions with $r_s^{(0)} < \rho$, the limited observed progress suggests that the current solution pattern may be inadequate, motivating $\textsc{Reformulate}$ to seek a more tractable solving path.

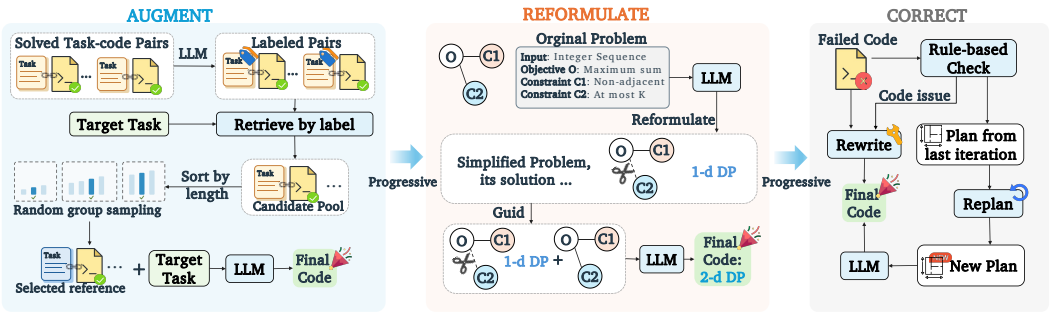


Figure 3: State-adaptive solving mechanisms in SACP-Coder: retrieval augmentation with candidate filtering (AUGMENT), reformulation-guided solution generation (REFORMULATE), and dual-level correction through REWRITE or REPLAN (CORRECT).

Feedback-Guided Iteration. To progressively refine solutions that remain unresolved after the current action, SACP-Coder iteratively refreshes the solver state with newly observed execution feedback and updates the subsequent action accordingly. After each non-terminal action, the updated candidate solution is re-executed on the public sample tests $\mathcal{T}_s$ to obtain the refreshed solver state $z_s^{(t+1)}$. Given $z_s^{(t+1)}$ and the preceding action $a_s^{(t)}$, the next action is determined by

$$a_s^{(t+1)} = \begin{cases} \textsc{Retain}, & r_s^{(t+1)} = 1, \\ \textsc{Reformulate}, & r_s^{(t+1)} < 1,\ a_s^{(t)} = \textsc{Augment}, \\ \textsc{Correct}, & r_s^{(t+1)} < 1,\ a_s^{(t)} \in \{\textsc{Reformulate}, \textsc{Correct}\}. \end{cases} \tag{8}$$

Accordingly, SACP-Coder terminates with RETAIN once the updated solution passes all available public sample tests. If AUGMENT remains insufficient, the solving process proceeds to REFORMULATE. Unresolved solutions after reformulation are further refined through CORRECT using the latest execution feedback.

3.3 STATE-ADAPTIVE SOLVING MECHANISMS

Retrieval Augmentation with Candidate Filtering. For problems assigned to AUGMENT, SACP-Coder leverages previously solved problems and their successful solutions as additional references for further solving. To obtain informative references from historical solutions, we design a retrieval augmentation strategy with candidate filtering, which retrieves related solved cases and further filters the retrieved candidates before incorporating them into the solving context. Specifically, for each previously solved problem s_j and its successful solution f_j, the LLM generates a set of problem-type labels $\mathcal{L}_j$. SACP-Coder organizes these labeled cases into a case bank:

$$\mathcal{B} = \{(s_j, f_j, \mathcal{L}_j)\}_{j=1}^{|\mathcal{B}|}. \tag{9}$$

Given a target problem s, the LLM generates its problem-type labels $\mathcal{L}_s$ in the same manner. Based on these labels, SACP-Coder retrieves up to N related solved cases from $\mathcal{B}$ to form a candidate pool $\mathcal{C}_s$. When fewer than N related cases are available, all available cases are retained. SACP-Coder then filters the retrieved candidates according to the code lengths of their solutions. The cases in $\mathcal{C}_s$ are sorted by solution length and divided into K approximately equal-sized groups, where K denotes the number of references to be selected. One case is randomly sampled from each group to form the final reference set $\mathcal{E}_s$. The selected problem-solution pairs are inserted into the context as references, and the solver agent regenerates the candidate solution as

$$f_s^{(t+1)} = M(s, \mathcal{E}_s). \tag{10}$$

Reformulation-Guided Solution Generation. For problems assigned to REFORMULATE, the current candidate may still rely on an ineffective solution pattern. SACP-Coder therefore reformulates the original problem into a simpler problem that preserves its core algorithmic structure, and uses the resulting guidance to reconstruct a solution for the original problem. Concretely, given a target problem s, the reformulation agent constructs the reformulation guidance

$$g_s = (\tilde{s}, k_s, u_s), \tag{11}$$

where $\tilde{s}$ denotes a simplified problem that preserves the core algorithmic structure of s, k_s captures its key solution pattern, and u_s characterizes the key differences between $\tilde{s}$ and s and specifies how this solution pattern should be adapted to solve the original problem. The solver agent then uses the reformulation guidance together with the original problem to generate a new candidate solution:

$$f_s^{(t+1)} = M(s, g_s)\,. \tag{12}$$

Dual-Level Correction. For problems assigned to CORRECT, applying the same correction to every failed solution may either discard a useful solution plan or preserve an ineffective one. To address this, we design a dual-level correction strategy that distinguishes different failure types and applies corrections at different levels accordingly. To determine the appropriate correction level, SACP-Coder first examines the current candidate $f_s^{(t)}$ through compilation checks and lightweight rule-based textual matching against the requirements of s, extracting directly identifiable code-level issues as the correction signal $d_s^{(t)} = \mathcal{D}(f_s^{(t)}, s)$. The correction level is then determined by

$$c_s^{(t)} = \begin{cases} \textsc{Rewrite}, & d_s^{(t)} \neq \varnothing, \\ \textsc{RePlan}, & d_s^{(t)} = \varnothing. \end{cases} \tag{13}$$

For REWRITE, SACP-Coder keeps the existing reformulation guidance g_s unchanged and directly revises the current candidate according to the detected issues and the requirements of the original problem:

$$f_s^{(t+1)} = M\left(s, f_s^{(t)}, d_s^{(t)}\right).$$

For REPLAN, SACP-Coder revises the existing reformulation guidance according to the observed failure. Given the original problem s, the previous guidance g_s, the failed candidate $f_s^{(t)}$, and the latest execution feedback $\phi_s^{(t)}$, the replanning agent revisits the existing guidance, identifies deficiencies in adapting the simplified problem to the original one, and accordingly revises the solution plan and its key algorithmic steps as $\hat{g}_s^{(t)} = \mathcal{P}(s, g_s, f_s^{(t)}, \phi_s^{(t)})$. The solver agent then generates the next candidate solution from the revised guidance:

$$f_s^{(t+1)} = M\left(s, \hat{g}_s^{(t)}\right).$$

4 EXPERIMENTAL EVALUATION

4.1 EXPERIMENTAL SETUP

Datasets and Metrics. We evaluate on four Python programming benchmarks: CodeContests (Li et al., 2022), APPS (Hendrycks et al., 2021), KodCode-OJ-Mix (Xu et al., 2025), and xCodeEval (Khan et al., 2024). We report accuracy (Acc., Pass@1, %), the percentage of final programs passing all evaluation tests, and average token consumption per problem (Tok., K), including all prompt and completion tokens across the entire solving process. Dataset details and the evaluation protocol appear in Appendix A.1.

Baselines. Alongside **Direct**, we compare three baseline groups: **Single-Agent Reasoning**, including CoT (Wei et al., 2022), StructCoT (Li et al., 2025b), and SelfPlanning (Jiang et al., 2024); **Multi-Agent Collaboration**, including MavenCoder (Xu et al., 2026), CodeTreeLite (Li et al., 2025c), PairCoder (Zhang et al., 2024), and MapCoder (Islam et al., 2024); and **Adaptive Computation**, including RoutingGen (Li et al., 2026), AdaCoder (Zhu et al., 2025), and PaT (Yoon et al., 2026). Implementation details appear in Appendix A.3.

Implementation Details. We evaluate SACP-Coder with DeepSeek-V4-Pro, Qwen3-Coder-30B, and GLM-4.5-Air; deployment and decoding settings appear in Appendix A.2. Retrieval uses $N = 10$ candidates and $K = 3$ references. We set $\rho = 0.5$ through validation-set calibration (Eq. 6); Appendix B.1 examines its robustness. By default, $R = 1$ limits the number of action updates after the initial mapping (Eq. 8). With $R = 0$, only the action selected by the initial mapping in Eq. 7 is executed. Further implementation details and prompt templates appear in Appendix A.4 and Appendix C, respectively.

Table 1: Comparison of SACP-Coder (Ours) with baselines using DeepSeek-V4-Pro. Micro Avg. aggregates results across the four benchmarks by problem. ▲ and ♦ mark the selected accuracy-nearest and token-nearest baselines to Ours, respectively. The bottom two rows show accuracy differences (pp) and relative token changes (%) against these references.

		CodeContest		KodCode		APPS		xCodeEval		Micro Avg.	
Category	Method	Acc. ↑	Tok. ↓	Acc. ↑	Tok. ↓	Acc. ↑	Tok. ↓	Acc. ↑	Tok. ↓	Acc. ↑	Tok. ↓
	Direct	41.82	1.01	74.67	0.75	67.33	0.78	58.49	0.80	60.25	0.84
Single-Agent Reasoning	CoT	38.79	4.06	73.33	0.95	71.33	2.83	52.83♦	5.94♦	59.02	3.27
	StructCoT	50.91	5.23	72.00	2.97	76.67	3.96	67.92	4.78	66.38	4.22
	SelfPlanning	50.91	5.49	72.00	2.83	80.00	4.02	69.81	5.36	67.60♦	4.38♦
Multi-Agent Collaboration	MavenCoder	57.58	16.44	80.67	5.08	78.00	8.78	71.70	31.56	71.63	14.25
	CodeTreeLite	60.61	17.20	82.67	5.16	80.67	12.15	65.09	43.90	72.51	17.67
	PairCoder	61.21	26.63	81.33	14.35	81.33	31.43	66.98	83.87	72.85	35.29
	MapCoder	61.82▲	32.24▲	82.67	13.89	82.00	16.79	73.58	21.87	74.78▲	21.44▲
Adaptive Computation	RoutingGen	35.76	3.39	74.67♦	1.87♦	74.67	2.87	62.26	4.93	61.12	3.14
	AdaCoder	56.36♦	10.92♦	83.33▲	1.65▲	76.00♦	3.80♦	67.92	23.76	70.75	9.00
	PaT-N3	56.36	39.25	77.33	4.94	82.00▲	14.77▲	73.58▲	11.91▲	71.80	18.73
Ours		**61.82**	8.69	**85.33**	1.94	**83.33**	3.50	**73.58**	5.88	**75.83**	**5.03**
vs. Acc.-nearest		(+0.00 pp)	(↓ 73.0%)	(+2.00 pp)	(↑ 17.6%)	(+1.33 pp)	(↓ 76.3%)	(+0.00 pp)	(↓ 50.6%)	(+1.05 pp)	(↓ 76.5%)
vs. Token-nearest		(+5.46 pp)	(↓ 20.4%)	(+10.66 pp)	(↑ 3.7%)	(+7.33 pp)	(↓ 7.9%)	(+20.75 pp)	(↓ 1.0%)	(+8.23 pp)	(↑ 14.8%)

Table 2: Comparison of allocation signals with fixed solving mechanisms ($R = 0$). Gray subscripts show Tok. Mis. reports each mismatch type as a percentage of all final failures under the corresponding signal. For Hard → Weak, a stronger branch that solves the problem is available, yet the signal selects a weaker branch that fails. For Easy → Strong , a weaker branch that solves the problem is available, yet the signal selects a stronger branch that fails. Hidden-test outcomes are used only for post-hoc analysis.

		CodeContest		KodCode		APPS		xCodeEval		Micro Avg.	
Signal	Mis. Type	Acc. ↑	Mis. ↓	Acc. ↑	Mis. ↓	Acc. ↑	Mis. ↓	Acc. ↑	Mis. ↓	Acc. ↑	Mis. ↓
Solv. State	Hard → Weak	**53.94**$_{4.45}$	**9.21**	**80.00**$_{1.46}$	**6.67**	**78.67**$_{2.13}$	**37.50**	**66.98**$_{2.79}$	**11.43**	**69.70**$_{2.75}$	**14.45**
	Easy → Strong		**9.21**		**0.00**		**0.00**		**2.86**		**4.62**
LLM Diff.	Hard → Weak	45.45$_{5.26}$	11.11	74.67$_{1.72}$	21.05	72.00$_{2.94}$	40.48	59.43$_{3.81}$	13.95	62.70$_{3.45}$	19.25
	Easy → Strong		20.00		5.26		11.90		16.28		15.02
Bench. Diff.	Hard → Weak	41.82$_{1.01}$	35.42	69.33$_{1.97}$	13.04	68.67$_{2.69}$	31.91	58.49$_{3.19}$	11.36	59.19$_{2.11}$	25.75
	Easy → Strong		0.00		26.09		25.53		20.45		14.16

4.2 Overall Effectiveness and Efficiency

To evaluate the effectiveness and efficiency of SACP-Coder, Table 1 compares its accuracy and token consumption with representative baselines across four benchmarks using DeepSeek-V4-Pro.

Three key findings demonstrate that SACP-Coder achieves a favorable performance-cost trade-off. **First**, SACP-Coder ranks first on KodCode and APPS and matches the best baseline accuracy on CodeContests and xCodeEval, achieving the highest micro-average accuracy of 75.83%. **Second**, compared with the accuracy-nearest baseline, MapCoder, on the micro-average results, SACP-Coder improves accuracy by 1.05 percentage points while reducing token consumption by 76.5%. Token savings hold on three of the four benchmarks against their respective accuracy-nearest baselines; on KodCode, it improves accuracy over AdaCoder by 2.00 percentage points with 17.6% more tokens. **Third**, across benchmarks, SACP-Coder outperforms the token-nearest baselines by 5.46–20.75 percentage points, with token costs ranging from 20.4% lower to 3.7% higher.

4.3 Effectiveness of Solver-State Guidance

To evaluate the effectiveness of solver state in state-conditioned action mapping, Table 2 compares three allocation signals with fixed solving mechanisms ($R = 0$): our solver state, difficulty esti-

Table 3: Cumulative ablation of SACP-Coder using DeepSeek-V4-Pro. Components are added cumulatively through the default configuration (+ Correction, $R = 1$); the last two rows increase the action-update limit R. Iteration enables only the AUGMENT-to-REFORMULATE transition, without CORRECT. Superscripts show changes in Acc. (pp) and Tok. (K) from the preceding row.

Cumulative Configuration	CodeContest		KodCode		APPS		xCodeEval	
	Acc. ↑	Tok. ↓	Acc. ↑	Tok. ↓	Acc. ↑	Tok. ↓	Acc. ↑	Tok. ↓
Direct	41.82	1.01	74.67	0.75	67.33	0.78	58.49	0.80
+ Augment	54.55$^{(+12.73)}$	3.56$^{(+2.55)}$	74.67$^{(+0.00)}$	1.16$^{(+0.41)}$	73.33$^{(+6.00)}$	1.96$^{(+1.18)}$	64.15$^{(+5.66)}$	2.99$^{(+2.19)}$
+ Reformulate	53.94$^{(-0.61)}$	4.45$^{(+0.89)}$	80.00$^{(+5.33)}$	1.46$^{(+0.30)}$	78.67$^{(+5.34)}$	2.13$^{(+0.17)}$	66.98$^{(+2.83)}$	2.79$^{(-0.20)}$
+ Iteration ($R = 1$)	54.55$^{(+0.61)}$	4.80$^{(+0.35)}$	80.00$^{(+0.00)}$	1.46$^{(+0.00)}$	81.33$^{(+2.66)}$	2.42$^{(+0.29)}$	66.98$^{(+0.00)}$	3.31$^{(+0.52)}$
+ Correction ($R = 1$)	**61.82**$^{(+7.27)}$	**8.69**$^{(+3.89)}$	**85.33**$^{(+5.33)}$	**1.94**$^{(+0.48)}$	**83.33**$^{(+2.00)}$	**3.50**$^{(+1.08)}$	**73.58**$^{(+6.60)}$	**5.88**$^{(+2.57)}$
↪ Correction ($R = 2$)	68.48$^{(+6.66)}$	14.29$^{(+5.60)}$	85.33$^{(+0.00)}$	2.04$^{(+0.10)}$	83.33$^{(+0.00)}$	6.11$^{(+2.61)}$	73.58$^{(+0.00)}$	10.52$^{(+4.64)}$
↪ Correction ($R = 3$)	68.48$^{(+0.00)}$	19.20$^{(+4.91)}$	85.33$^{(+0.00)}$	2.16$^{(+0.12)}$	84.67$^{(+1.34)}$	10.02$^{(+3.91)}$	73.58$^{(+0.00)}$	18.30$^{(+7.78)}$

mated by the base LLM from the problem statement (LLM Diff.), and difficulty annotated in the benchmark (Bench. Diff.). The two difficulty-based signals map easy, medium, and hard labels to RETAIN, AUGMENT, and REFORMULATE, respectively, while solver-state guidance follows Eq. 7. First, solver-state guidance achieves the highest accuracy on all four benchmarks while using fewer tokens than LLM-estimated difficulty, demonstrating its effectiveness in selecting computation actions. To further understand the performance gaps, we analyze failed solutions under each signal and identify two types of allocation mismatch. Hard → Weak indicates cases where a stronger branch successfully solves the problem, but the signal selects a weaker branch whose solution fails. Similarly, Easy → Strong indicates cases where a weaker branch already solves the problem, but the signal selects a stronger branch whose solution fails instead. At the micro-average level, both mismatch types account for larger shares of final failures under two difficulty-based signals, helping explain their lower accuracy. Solver-state guidance reduces both types of mismatch, better aligning computation actions with current solving needs.

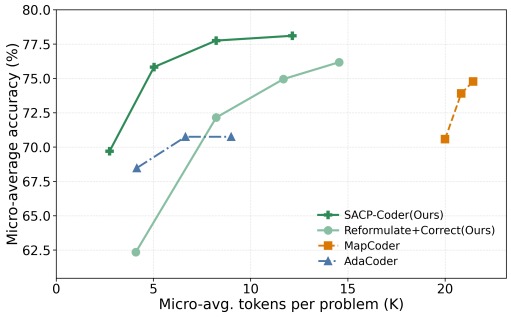
4.4 ABALATION STUDY

To examine the contributions of individual mechanisms, Table 3 cumulatively adds state-adaptive solving mechanisms and feedback-guided iteration to Direct generation. **First**, AUGMENT improves accuracy on three benchmarks, and adding REFORMULATE yields further gains on three, with a slight drop on CodeContests potentially reflecting cases reassigned from AUGMENT to REFORMULATE before CORRECT is enabled. These complementary gains motivate providing the two mechanisms through state-conditioned action mapping. **Second**, enabling only the AUGMENT-to-REFORMULATE transition improves accuracy on CodeContests and APPS, which implies feedback-guided iteration can recover solutions that remain unresolved after retrieval augmentation, even without CORRECT. **Third**, adding dual-level correction improves accuracy on all four benchmarks by 2.00–7.27 percentage points. This highlights the value of revising candidate code or reformulation guidance after problem reformulation remains unsuccessful. Finally, increasing R brings limited additional accuracy gains at higher token costs, supporting $R = 1$ as an economical default.

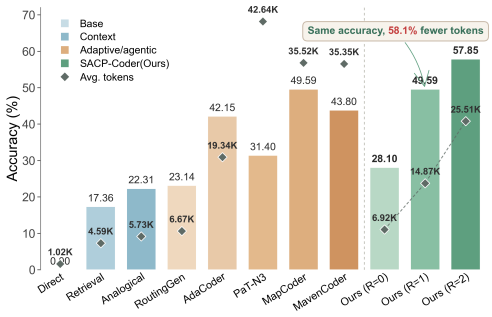
5 FURTHER ANALYSIS

5.1 PROGRESSIVE TEST-TIME COMPUTATION SCALING

To assess how additional computation translates into solving performance, Fig. 4a varies each method's iteration limit and reports accuracy against token consumption, pooling problems across benchmarks. First, SACP-Coder forms the empirical Pareto frontier among the evaluated configurations and achieves the highest overall accuracy at lower token cost than the strongest baseline configurations. Its performance-cost advantage therefore extends beyond the default iteration limit. Second, SACP-Coder reaches comparable or higher accuracy with fewer tokens than uniformly applying REFORMULATE and CORRECT. This comparison highlights the value of state-conditioned action mapping and feedback-guided iteration in aligning computation with current solving needs. Per-benchmark curves and iteration settings are provided in Appendix B.2.



(a) Progressive test-time computation scaling.



(b) Solving challenging problems.

Figure 4: (a) Accuracy-token trade-offs as iteration limits increase. Reformulate+Correct applies problem reformulation to every problem, followed by up to three correction rounds. (b) Results on problems with $r_s^{(0)} = 0$. Bars show Acc.; diamonds show Tok. SACP-Coder starts with REFORMULATE and allows up to R subsequent CORRECT operations.

5.2 SOLVING CHALLENGING PROBLEMS

To evaluate performance on challenging problems, Fig. 4b compares methods on the subset of problems with a Direct pass ratio of $r_s^{(0)} = 0$. We additionally compare Retrieval, which retrieves three reference examples based on problem-type labels, and Analogical, which prompts the LLM to generate three reference examples. First, REFORMULATE alone ($R = 0$) outperforms retrieved and generated analogical examples, while adding CORRECT ($R = 1$) matches MapCoder's 49.59% accuracy with 58.1% fewer tokens. Reformulation-guided solution generation, together with dual-level correction, thus provides an effective and economical solving path for these challenging problems. Second, increasing R further raises accuracy above all compared baselines, showing that additional dual-level correction can recover more solutions after problem reformulation. To illustrate how a simpler problem preserving the core algorithmic structure guides solution generation for the original problem, we present a case study in Appendix B.3.

5.3 RESULTS ACROSS LLM BACKBONES

To evaluate SACP-Coder across LLM backbones, Table 4 reports micro-average results for Qwen3-Coder-30B and GLM-4.5-Air; full per-benchmark comparisons are provided in Appendix B.4 (Table 6). First, SACP-Coder achieves the highest micro-average accuracy among the compared methods on both backbones. Second, it reduces token consumption relative to the accuracy-nearest baseline, MapCoder, by 76.1% and 78.4%, respectively, while achieving slightly higher accuracy. Together with the DeepSeek-V4-Pro results, these findings show that SACP-Coder maintains a favorable performance-cost trade-off across the three LLM backbones.

Table 4: Micro-averaged results on additional LLM backbones. Notation and markers follow Table 1.

		Qwen3-Coder-30B		GLM-4.5-Air	
Category	Method	Acc. ↑	Tok. ↓	Acc. ↑	Tok. ↓
	Direct	39.93	3.08	47.99	2.61
Single Agent	CoT	38.53	4.45	45.18	3.89
Multi-Agent	MavenCoder	40.63	16.28	51.31	16.73
	MapCoder	47.29▲	36.79▲	54.47▲	34.32▲
Adaptive	RoutingGen	40.28◆	4.82◆	46.41◆	4.14◆
	AdaCoder	43.08	13.27	50.72	11.08
	PaT-N3	35.20	25.12	36.95	32.88
Ours		**47.99**	8.80	**54.99**	7.42
vs. Acc.-nearest		(+0.70 pp)	(↓ 76.1%)	(+0.52 pp)	(↓ 78.4%)
vs. Token-nearest		(+7.71 pp)	(↑ 82.7%)	(+8.58 pp)	(↑ 79.3%)

6 CONCLUSION

In this paper, we presented SACP-Coder, which adapts inference-time computation to the evolving solver state through retrieval augmentation, problem reformulation, and dual-level correction. In particular, reformulation derives a simpler problem that preserves the core algorithmic structure and guides the transfer of its solution pattern to the original problem. Experiments across four benchmarks and three LLM backbones demonstrate a favorable performance-cost trade-off, supporting the value of aligning computation with current solving needs.

AI USE STATEMENT

Generative AI tools were used only to assist with code debugging and to improve the clarity, grammar, and readability of the manuscript. They were not used for research ideation, hypothesis formulation, methodology or experimental design, data generation or processing, mathematical claims or proofs, or the analysis or interpretation of results. All AI-assisted debugging suggestions were reviewed and tested by the authors, and all language edits were manually verified. The methods, experiments, and scientific conclusions were independently developed and validated by the authors, who take full responsibility for the final content of this work.

ETHICS STATEMENT

This work studies inference-time computation allocation for LLM-based code generation using established programming benchmarks and does not involve human subjects or the collection of new personal or sensitive data. The proposed framework can reduce token consumption by avoiding unnecessary model calls, potentially lowering inference cost and resource use. However, generated programs may contain functional errors, security vulnerabilities, or harmful behavior, and passing public sample tests does not guarantee correctness on unseen inputs or safety in real-world settings. Improvements in the capability and efficiency of code generation could also facilitate misuse. Generated code should therefore be executed in isolated environments and undergo independent testing and human review before deployment. The framework should be used in accordance with the licenses and terms of the underlying models and datasets, with appropriate safety and access controls.

REPRODUCIBILITY STATEMENT

To support the reproducibility of our experimental results, we provide a detailed description of the proposed method together with comprehensive documentation of the experimental setup and method-specific parameters. Section 3 formally specifies the solver state, state-conditioned action mapping, retrieval augmentation, problem reformulation, and dual-level correction mechanisms. Appendix A documents the benchmark composition, evaluation protocol, model and inference settings, baseline implementations, and method-specific hyperparameters. Appendix B provides additional per-benchmark analyses, while Appendix C includes the complete prompt templates used by SACP-Coder.

REFERENCES

Pranjal Aggarwal, Aman Madaan, Yiming Yang, and Mausam. Let's sample step by step: Adaptive-consistency for efficient reasoning and coding with LLMs. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 12375–12396, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.761. URL https://aclanthology.org/2023.emnlp-main.761/.

Lingjiao Chen, Matei Zaharia, and James Zou. FrugalGPT: How to use large language models while reducing cost and improving performance. *Transactions on Machine Learning Research*, 2024a. ISSN 2835-8856. URL https://openreview.net/forum?id=cSimKw5p6R. Featured Certification.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *The Twelfth International Conference on Learning Representations*, 2024b. URL https://openreview.net/forum?id=KuPixIqPiq.

Mehul Damani, Idan Shenfeld, Andi Peng, Andreea Bobu, and Jacob Andreas. Learning how hard to think: Input-adaptive allocation of LM computation. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=6qUUgw9bAZ.

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

Tatsunori B. Hashimoto, Kelvin Guu, Yonatan Oren, and Percy Liang. A retrieve-and-edit framework for predicting structured outputs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, pp. 10073–10083, Red Hook, NY, USA, 2018. Curran Associates Inc.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps. *NeurIPS*, 2021.

Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 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)*, pp. 4912–4944, 2024.

Geonhui Jang, Dongyoon Han, and YoungJoon Yoo. StoryCoder: Narrative reformulation for structured reasoning in LLM code generation. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 19092–19112, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.872. URL https://aclanthology.org/2026.acl-long.872/.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*, 35 (2):1–72, 2026.

Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. Self-planning code generation with large language models. *ACM Trans. Softw. Eng. Methodol.*, 33(7), September 2024. ISSN 1049-331X. doi: 10.1145/3672456. URL https://doi.org/10.1145/3672456.

Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. XCodeEval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 6766–6805, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.367. URL https://aclanthology.org/2024.acl-long.367/.

Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq Joty. Codechain: Towards modular code generation through chain of self-revisions with representative sub-modules. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=vYhglxSj8j.

Dacheng Li, Shiyi Cao, Chengkun Cao, Xiuyu Li, Shangyin Tan, Kurt Keutzer, Jiarong Xing, Joseph E. Gonzalez, and Ion Stoica. S*: Test time scaling for code generation. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 15964–15978, Suzhou, China, November 2025a. Association for Computational Linguistics. ISBN 979-8-89176-335-7. doi: 10.18653/v1/2025.findings-emnlp.865. URL https://aclanthology.org/2025.findings-emnlp.865/.

Jia Li, Ge Li, Yongmin Li, and Zhi Jin. Structured chain-of-thought prompting for code generation. *ACM Trans. Softw. Eng. Methodol.*, 34(2), January 2025b. ISSN 1049-331X. doi: 10.1145/3690635. URL https://doi.org/10.1145/3690635.

Jierui Li, Hung Le, Yingbo Zhou, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. CodeTree: Agent-guided tree search for code generation with large language models. In Luis Chiruzzo, Alan Ritter, and Lu Wang (eds.), *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)*, pp. 3711–3726, Albuquerque, New Mexico, April 2025c. Association for Computational Linguistics. ISBN 979-8-89176-189-6. doi: 10.18653/v1/2025.naacl-long.189. URL https://aclanthology.org/2025.naacl-long.189/.

Shen Li, Li Huang, Shaoxiong Zhan, Weifeng Sun, Tao Yin, Zhongxin Liu, and Meng Yan. Intention chain-of-thought prompting with dynamic routing for code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 650–658, 2026.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022. doi: 10.1126/science.abq1158.

Charlie Victor Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling parameters for reasoning. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=4FWAwZtd2n.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc. ISBN 9781713871088.

Yuyang Wu, Yifei Wang, Ziyu Ye, Tianqi Du, Stefanie Jegelka, and Yisen Wang. When more is less: Understanding chain-of-thought length in LLMs. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=6QDFsYxtI1.

Zhangchen Xu, Yang Liu, Yueqin Yin, Mingyuan Zhou, and Radha Poovendran. KodCode: A diverse, challenging, and verifiable synthetic dataset for coding. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 6980–7008, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.365. URL https://aclanthology.org/2025.findings-acl.365/.

ZhenChun Xu, Yi Cai, Dajun Zheng, Li Yuan, Mengchen Zhao, Qixiang Wang, and Jiexin Wang. Mavencoder: Competitive code generation via model adaptive planning strategies and multi-perspective verification enhancement. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 30415–30439, 2026.

Michihiro Yasunaga, Xinyun Chen, Yujia Li, Panupong Pasupat, Jure Leskovec, Percy Liang, Ed H. Chi, and Denny Zhou. Large language models as analogical reasoners. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=AgDICX1h50.

Youngsik Yoon, Sungjae Lee, Seockbean Song, Siwei Wang, Wei Chen, and Jungseul Ok. PaT: Planning-after-trial for efficient test-time code generation. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 36738–36755, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.1703. URL https://aclanthology.org/2026.acl-long.1703/.

Huan Zhang, Wei Cheng, Yuhan Wu, and Wei Hu. 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*, ASE '24, pp. 1319–1331, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400712487. doi: 10.1145/3691620.3695506. URL https://doi.org/10.1145/3691620.3695506.

Yueheng Zhu, Chao Liu, Xuan He, Xiaoxue Ren, Zhongxin Liu, Ruwei Pan, and Hongyu Zhang. Adacoder: An adaptive planning and multi-agent framework for function-level code generation. *IEEE Transactions on Software Engineering*, 2025.

A Additional Experimental Details

A.1 Datasets and Evaluation Protocol

We evaluate on four benchmarks: **CodeContests** (Li et al., 2022) (165 problems), comprising competitive-programming tasks with standard-input/standard-output interfaces; **APPS** (Hendrycks et al., 2021) (150 problems), covering introductory, interview, and competition levels; **KodCode-OJ-Mix** (Xu et al., 2025) (150 problems), combining online-judge tasks from five source collections; and **xCodeEval** (Khan et al., 2024) (106 problems), providing competitive-programming tasks for program synthesis. For threshold calibration, we additionally sample a validation set from each benchmark's source dataset, with a size equal to 10% of its evaluation set and no overlap with the evaluated problems.

Each evaluated problem includes public sample tests and hidden evaluation tests. Across all methods, execution feedback during solution generation comes exclusively from public sample tests. Hidden tests are used only to evaluate the final returned program and never guide generation, correction, or candidate selection.

A.2 Models and Inference Settings

We evaluate SACP-Coder with DeepSeek-V4-Pro,[1] Qwen3-Coder-30B-A3B-Instruct,[2] and GLM-4.5-Air.[3] DeepSeek-V4-Pro is accessed through its official API. Qwen3-Coder-30B-A3B-Instruct and GLM-4.5-Air are deployed locally with vLLM on one and four Huawei Ascend 910C accelerators, respectively, each with 128 GB of device memory. All methods use top-p 0.95 and a maximum of 8,192 output tokens per call, with thinking mode disabled. Temperature is 0 except for PaT trial generation, which uses different temperatures across candidates.

A.3 Baseline Implementations

We compare against three categories of baselines spanning established approaches and recent advances. We use the authors' released code wherever possible and retain their original method-specific settings with minimal changes, subject to the shared inference settings above. We also include **Direct**, which generates a solution in 1 model call.

Single-Agent Reasoning. These methods introduce explicit reasoning or planning within a single model. **CoT** (Wei et al., 2022) generates step-by-step reasoning followed by code in 1 call. **Struct-CoT** (Li et al., 2025b) first constructs a structured reasoning trace and then generates code, using 2 calls per problem. **SelfPlanning** (Jiang et al., 2024) generates an ordered solution plan and then implements it, using 2 calls per problem.

Multi-Agent Collaboration. These methods combine specialized roles for planning, candidate exploration, verification, and debugging. **MavenCoder** (Xu et al., 2026) selects a difficulty-dependent planning strategy and verifies and debugs the resulting solution. **CodeTreeLite** (Li et al., 2025c) is our bounded implementation of CodeTree-style search, in which Thinker, Solver, Critic, and Debugger roles explore and refine candidate solutions. **PairCoder** (Zhang et al., 2024) pairs a Navigator that proposes and switches plans with a Driver that generates and repairs code. **MapCoder** (Islam et al., 2024) recalls relevant problems, ranks their associated plans, and performs coding and debugging. By default, we use 3 recalled problems and allow up to 3 debugging rounds per attempted plan.

Adaptive Computation. These methods adjust reasoning, planning, or refinement computation according to problem difficulty or intermediate solving outcomes. **RoutingGen** (Li et al., 2026) classifies problems as simple or complex and routes them to direct generation or intention-based planning followed by code generation. We use automatic routing with the same base model. **AdaCoder** (Zhu et al., 2025) starts with direct generation and invokes feedback-aware planning and

[1] https://www.deepseek.com/platform/

[2] https://huggingface.co/Qwen/Qwen3-Coder-30B-A3B-Instruct

[3] https://huggingface.co/zai-org/GLM-4.5-Air

regeneration after execution failures. By default, we allow up to 3 rounds of corrections. **PaT** (Yoon et al., 2026) generates trial programs and invokes recursive planning if none succeeds. We use 3 trial candidates generated at different temperatures (PaT-N3).

A.4 SACP-CODER IMPLEMENTATION DETAILS

Retrieval Augmentation with Candidate Filtering. The case bank contains only previously generated and successfully verified problem–solution pairs from the same dataset as the target problem. The LLM assigns problem-type labels to both stored cases and the target, and retrieval uses these labels to identify related cases. We retrieve up to $N = 10$ candidates, retaining all available related cases if fewer than ten are available. Candidates are sorted by solution length and divided into approximately equal-sized groups; with ten candidates and $K = 3$, the group sizes are 3, 4, and 3. One case is randomly selected from each group as a reference for AUGMENT.

Feedback-Guided Iteration. The parameter R limits the number of action updates under Eq. 8, starting after the action selected by the initial mapping in Eq. 7; neither the initial **Direct** generation nor the initially selected action counts toward R. Thus, $R = 0$ executes only the initially selected action, while $R = 1, 2, 3$ permit at most one, two, or three subsequent updates, respectively. For an initial REFORMULATE action, these updates allow up to R CORRECT operations; for an initial AUGMENT action, escalation to REFORMULATE consumes the first update, leaving at most $R - 1$ corrections when $R \geq 1$. Execution stops early with RETAIN once all public sample tests pass.

Dual-Level Correction. Within CORRECT, compilation checks and lightweight rule-based textual matching identify syntax errors or interface mismatches; detected issues trigger REWRITE, which preserves the existing reformulation guidance. If these checks identify no such issue despite failed public sample tests, REPLAN revises the reformulation guidance using execution feedback before generating a new candidate.

B ADDITIONAL RESULTS AND ANALYSIS

B.1 SENSITIVITY TO THE DECISION THRESHOLD

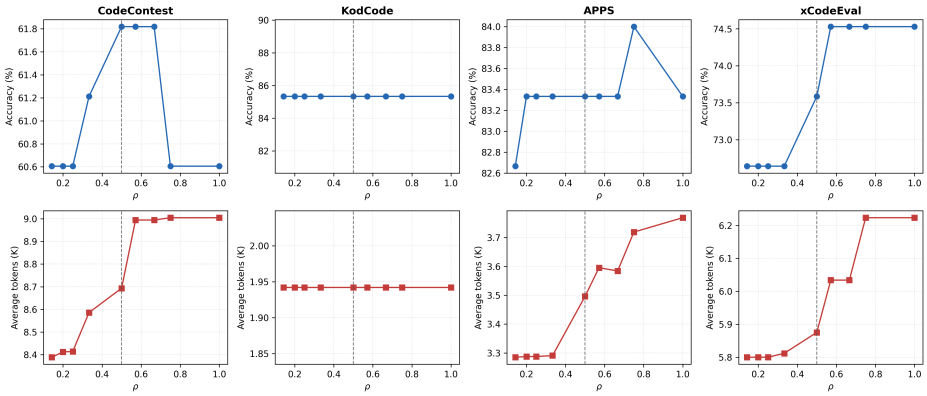
To assess the robustness of the decision threshold used in Section 4.1, Fig. 5 varies ρ under different limits on the number of action updates. Accuracy changes modestly across the tested thresholds, while token consumption reflects the cost of different initial action mappings. These results support the default threshold as a reasonable choice without requiring it to maximize accuracy on every benchmark.

B.2 PER-BENCHMARK COMPUTATION SCALING

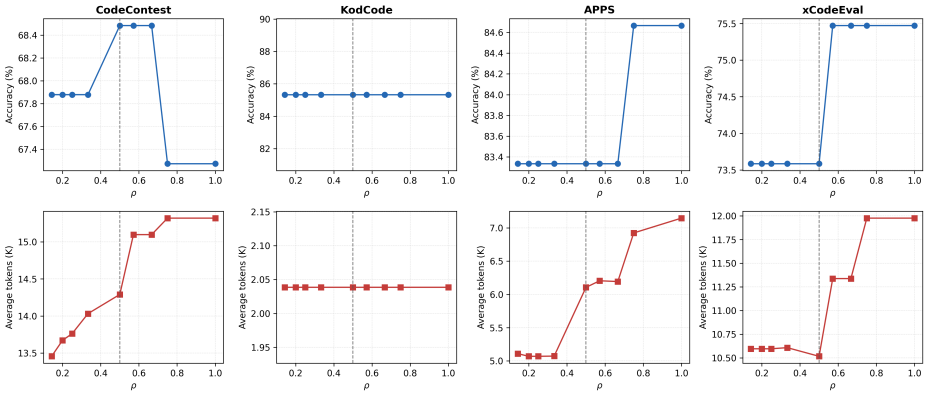
To examine whether the pooled scaling trend holds across benchmarks, Fig. 6 expands the comparison in Section 5.1 and Fig. 4a. SACP-Coder offers a favorable accuracy–token trade-off across benchmarks compared with the baselines and uniform REFORMULATE+CORRECT, supporting the value of state-conditioned action mapping and feedback-guided iteration. Gains from additional updates vary by benchmark and can plateau, so increased computation does not always improve accuracy.

B.3 CASE STUDY OF REFORMULATION-GUIDED SOLUTION GENERATION

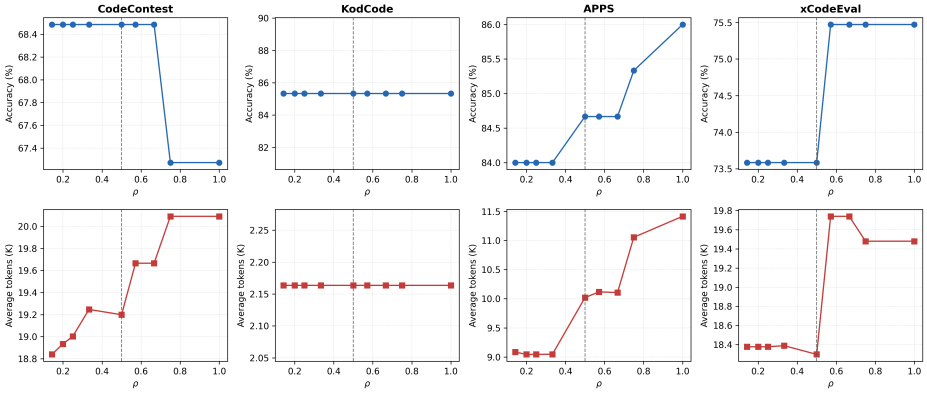
To illustrate how reformulation-guided solution generation helps solve a challenging problem, Table 5 complements the evaluation in Section 5.2 and Fig. 4b. Reformulating polynomial counting as a bounded power-of-two representation exposes a digit-DP solution pattern with consistent transitions and a valid stopping condition. In this case, the resulting program passes evaluation without CORRECT, illustrating how reformulation guidance can address the errors shown in the baseline solutions.



(a) $R = 1$.



(b) $R = 2$.



(c) $R = 3$.

Figure 5: Sensitivity to the decision threshold ρ under different limits R on the number of action updates. Each panel shows accuracy (top) and token consumption (bottom); dashed lines mark the default threshold.

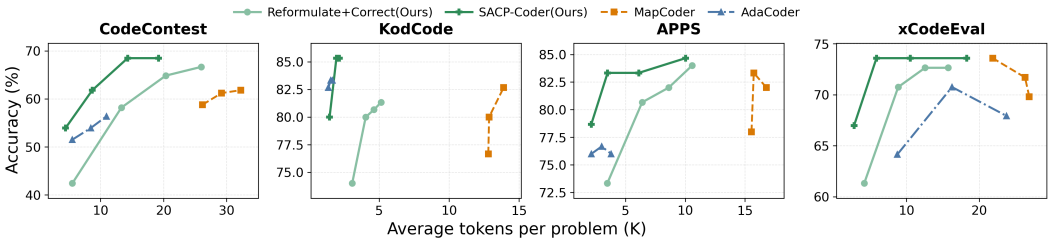


Figure 6: Per-benchmark accuracy–token trade-offs. SACP-Coder uses $R = 0, 1, 2, 3$; Reformulate+Correct applies REFORMULATE to every problem followed by up to zero, one, two, or three corrections. MapCoder and AdaCoder vary their debugging and correction limits, respectively, from one to three.

Table 5: Reformulation-guided solution generation on APPS #2233 without CORRECT. Blue panels show reformulation guidance; highlighted code compares transitions and stopping conditions.

Problem. Count polynomials $P(x) = \sum_i a_i x^i$ satisfying $P(2) = m$, where $a_i \in \{0, \ldots, 7\}$.
Example. For $m = 4$, the valid polynomials are 4, $2 + x$, $2x$, and x^2: **four solutions**.

Baseline failures

Direct: incorrect transition

```
r = m & 7
m >>= 3
...
val = coeff * 2 + carry
if val & 7 == r:
  new_dp[val >> 3] += ...
```

Error. Consumes three bits at once without accounting for the corresponding coefficient combinations.

MapCoder: incorrect stopping condition

```
while m > 0 or any(
  dp[i] for i in
  range(1, MAX_CARRY + 1)
):
  bit = m & 1
  ...
  m >>= 1
```

Error. Continues expanding invalid nonzero-carry states after all target bits are consumed, causing a **timeout** in the local replay.

Ours: reformulation-guided generation

Reformulated problem

"Count the number of ways to represent a given integer m as a sum of powers of 2, where each power of 2 can be used at most 7 times."

Key solution pattern

"This can be solved via digit DP on the binary representation of m, where the state tracks the current bit position and the carry from lower bits."

Generated code: bitwise transition

```
s = cin + a
if s % 2 == bit:
  cout = s // 2
  trans[bit][cin][cout] += 1
```

Generated code: stopping condition

```
while m > 0:
  bit = m & 1
  ...
  m >>= 1
return dp[0] % MOD
```

Accept only zero-carry states after the target bits have been processed.

Input	Expected	Direct	MapCoder	Ours
$m = 2$	2	1	Timeout†	**2**
$m = 4$	4	1	Timeout†	**4**

†The original MapCoder program exceeds a 60-second limit on each input in a local replay. Code excerpts omit unrelated lines and are wrapped for readability.

Outcome. Ours passes the recorded evaluation with its **initial code**, without CORRECT. AdaCoder also passes this case.

B.4 DETAILED RESULTS ACROSS LLM BACKBONES

To examine performance across benchmarks and LLM backbones, Table 6 provides the detailed results underlying Section 5.3 and Table 4. SACP-Coder improves on Direct across all evaluated

Table 6: Per-benchmark and micro-average results on Qwen3-Coder-30B-A3B-Instruct and GLM-4.5-Air; Ours denotes SACP-Coder. Acc. is Pass@1 (%); Tok. is average total prompt and completion tokens per problem (K). Micro Avg. aggregates results across benchmarks by problem. ▲ and ♦ mark the selected accuracy-nearest and token-nearest baselines to Ours, respectively. The final two rows in each model block report accuracy differences in percentage points (pp) and relative token changes (%) against these references; ↓/↑ indicate token reductions/increases.

Qwen3-Coder-30B

		CodeContests		KodCode		APPS		xCodeEval		Micro Avg.	
Category	Method	Acc. ↑	Tok. ↓	Acc. ↑	Tok. ↓	Acc. ↑	Tok. ↓	Acc. ↑	Tok. ↓	Acc. ↑	Tok. ↓
	Direct	13.94	4.88	64.00	1.33	45.33	2.43	38.68	3.67	39.93	3.08
Single-Agent Reasoning	CoT	16.36	6.29	62.67	1.26	42.00	3.75	33.96♦	7.10♦	38.53	4.45
Multi-Agent Collaboration	MavenCoder	18.79	25.61	63.33	5.74	46.00	12.39	34.91	22.18	40.63	16.28
	MapCoder	24.24▲	63.87▲	69.33▲	17.14▲	52.67	27.63	44.34	35.43	47.29▲	36.79▲
Adaptive Computation	RoutingGen	13.33	7.17	64.00	2.61	47.33	4.31	38.68	4.99	40.28♦	4.82♦
	AdaCoder	18.18♦	23.91♦	67.33♦	3.20♦	51.33▲♦	8.04▲♦	35.85	18.38	43.08	13.27
	PaT-N3	26.67	46.33	66.67	6.99	7.33	23.51	43.40▲	20.02▲	35.20	25.12
Ours		24.24	15.71	**74.67**	2.96	50.67	6.66	43.40	9.32	**47.99**	8.80
vs. Acc.-nearest		(+0.00 pp)	(↓ 75.4%)	(+5.34 pp)	(↓ 82.7%)	(-0.66 pp)	(↓ 17.2%)	(+0.00 pp)	(↓ 53.4%)	(+0.70 pp)	(↓ 76.1%)
vs. Token-nearest		(+6.06 pp)	(↓ 34.3%)	(+7.34 pp)	(↓ 7.5%)	(-0.66 pp)	(↓ 17.2%)	(+9.44 pp)	(↑ 31.3%)	(+7.71 pp)	(↑ 82.7%)

GLM-4.5-Air

		CodeContests		KodCode		APPS		xCodeEval		Micro Avg.	
Category	Method	Acc. ↑	Tok. ↓	Acc. ↑	Tok. ↓	Acc. ↑	Tok. ↓	Acc. ↑	Tok. ↓	Acc. ↑	Tok. ↓
	Direct	23.03	4.14	70.67	1.29	56.00	2.20	43.40	2.70	47.99	2.61
Single-Agent Reasoning	CoT	20.00	5.38	68.00	0.98	54.00	3.11	39.62♦	6.81♦	45.18	3.89
Multi-Agent Collaboration	MavenCoder	29.09	22.80	74.67▲	5.42▲	58.67	11.78	42.45	30.29	51.31	16.73
	MapCoder	31.52▲	55.25▲	74.00	16.73	66.00▲	28.55▲	46.23	34.78	54.47▲	34.32▲
Adaptive Computation	RoutingGen	21.82	6.30	72.00♦	2.35♦	48.00♦	3.73♦	46.23▲	3.88▲	46.41♦	4.14♦
	AdaCoder	31.52♦	16.45♦	73.33	2.98	58.00	7.67	38.30	19.02	50.72	11.08
	PaT-N3	22.42	52.79	70.00	8.31	16.67	33.59	41.51	35.64	36.95	32.88
Ours		**32.12**	13.32	**76.00**	2.58	64.67	5.57	**47.17**	7.71	**54.99**	7.42
vs. Acc.-nearest		(+0.60 pp)	(↓ 75.9%)	(+1.33 pp)	(↓ 52.4%)	(-1.33 pp)	(↓ 80.5%)	(+0.94 pp)	(↑ 98.7%)	(+0.52 pp)	(↓ 78.4%)
vs. Token-nearest		(+0.60 pp)	(↓ 19.0%)	(+4.00 pp)	(↑ 9.8%)	(+16.67 pp)	(↑ 49.3%)	(+7.55 pp)	(↑ 13.2%)	(+8.58 pp)	(↑ 79.3%)

benchmark–backbone combinations and achieves the highest micro-average accuracy among the compared methods on both backbones, with lower token consumption than MapCoder. Although it does not lead in accuracy on every benchmark, the overall results support its effectiveness and efficiency across backbones.

C PROMPT TEMPLATES

C.1 DIRECT SOLUTION GENERATION

Direct

```
[problem statement, input/output specification, and public examples]

Generate Python3 code to solve the above mentioned problem:
```

C.2 RETRIEVAL AUGMENTATION WITH CANDIDATE FILTERING

Augment: Solution Generation

```
[reference 1: retrieved problem statement]
```python3
[reference 1: retrieved correct solution]
```

[reference 2: retrieved problem statement]
```python3
[reference 2: retrieved correct solution]
```

[reference 3: retrieved problem statement]
```python3
[reference 3: retrieved correct solution]
```

[target problem statement, input/output specification, and public
    examples]

Let's think step by step and generate Python3 code to solve the
    problem.

[dataset-specific interface instruction]

Important: Your response must contain only the Python3 code to solve
    this problem inside ``` block.
```

C.3 REFORMULATION-GUIDED SOLUTION GENERATION

Reformulate: Reformulation Guidance

```
Given a competitive programming problem, do not write code yet.

Your task is to derive compact reformulation guidance.

A simplified problem is a simpler but homologous problem:
```

```
- It should be easier than the original problem.
- It must preserve the original problem's main algorithmic structure.
- It should not remove the decisive condition that makes the original
    problem what it is.
- It should help explain how the original problem evolves from a
    simpler known pattern.

# Original Problem:
[target problem]

## Sample Test cases:
[public examples]

# Target Interface Contract:
[required function or stdin/stdout contract]

Important:
- The simplified problem is an intermediate reasoning object, not the
    final problem.
- Prefer one clean simplified problem over many examples.
- The simplified problem should be simpler, but it must remain
    structurally related to the original problem.
- Do not choose a simplified problem that requires a completely
    different algorithm.
- Do not write final code in this response.
- Keep the plan concise and concrete.
- Your response must follow the XML schema exactly.
<root>
  <simplified_problem>[simpler homologous
    problem]</simplified_problem>
  <key_solution_pattern>[key observation and
    solution]</key_solution_pattern>
  <problem_delta>[additional conditions in the target]</problem_delta>
  <evolution_to_original>[how to adapt the simplified
    solution]</evolution_to_original>
  <boundary_checks>[correctness-critical boundary
    cases]</boundary_checks>
</root>
```

Reformulate: Solution Generation

```
Given the original competitive programming problem and reformulation
    guidance, generate Python3 code for the original problem.

# Original Problem To Solve
[target problem and public examples]

# Simplified Problem
[simplified_problem]

# Key Solution Pattern
Key idea and algorithm used in the simplified problem:

[key_solution_pattern]

# Added Constraints In The Original Problem
Constraints and complications that exist in the original problem but
    not in the simplified problem:

[problem_delta]

# Adapting the Solution Pattern to the Original Problem
```

```
Notes on adapting the simplified solution pattern toward the original
    problem:

[evolution_to_original]

# Boundary Cases For The Original Problem
Boundary cases that the final solution must handle:

[boundary_checks]

# Target Interface Contract
[required function or stdin/stdout contract]

Before writing the final code, use the reformulation guidance as a
    reference:
1. Identify which parts of the simplified problem's algorithm remain
    useful.
2. Identify which parts must change because of the added constraints.
3. Apply the necessary changes in the final algorithm.

# Final Output
Output only complete Python3 code for the original problem.

Do not include derivation comments.

Do not restate the reformulation guidance.
```

C.4 DUAL-LEVEL CORRECTION

Correct: Rewrite

```
The previous Python3 solution for this problem failed the public
    sample tests.

Your task is to repair the code once using the detected code-level
    issues.

# Original Problem
[target problem and public examples]

# Current Reformulation Guidance
[latest reformulation guidance used to generate the current code]

# Current Code That Failed Public Samples
[current code]

# Detected Code-Level Issues
[issues identified by compilation checks and lightweight rule-based
    textual matching]

# Public Sample Test Feedback
[passed and failed examples with observed outputs]

# Target Interface Contract
[required function or stdin/stdout contract]

Repair rules:
1. Do not hardcode the sample tests.
2. Prefer minimal fixes for parsing, output format, multi-test
    handling, off-by-one errors, boundary cases, and condition
    mistakes.
```

```
3. Keep the overall algorithm when it is still consistent with the
    reformulation guidance.
4. Change the algorithm only if the failed sample proves the
    algorithm is wrong.
5. Return only complete Python3 code. Do not add explanation or
    markdown fences.
```

Correct: RePlan – Revised Guidance

```
The current candidate solution failed the public sample tests.

Do not write code in this step. Re-plan how to solve the original
    problem.

# Original Problem
[target problem and public examples]

# Current Reformulation Guidance
[latest reformulation guidance used to generate the current code]

# Failed Code
[current code]

# Public Sample Test Feedback
[passed and failed examples with observed outputs]

# Target Interface Contract
[required function or stdin/stdout contract]

Replanning rules:
1. Diagnose whether the current reformulation guidance missed an
    original constraint, input/output detail, boundary case, or
    algorithmic requirement.
2. Produce a corrected transfer plan for the original problem, not a
    patch to the code.
3. Keep the plan compact and implementation-oriented.
4. Do not reference hidden tests.

Return XML only with the following fields: failure_diagnosis,
    reformulation_guidance_issue, revised_transfer_plan,
    revised_key_steps, revised_pseudocode, revised_complexity, and
    implementation_warnings.
```

Correct: RePlan – Solution Generation

```
Write the final Python3 solution for the original problem.

# Original Problem
[target problem and public examples]

# Current Reformulation Guidance
[latest reformulation guidance used to generate the current code]

# Revised Transfer Plan
[revised plan produced in this correction round]

# Target Interface Contract
[required function or stdin/stdout contract]

Coding rules:
```

```
1. Use the revised transfer plan as the primary plan.
2. Generate a complete solution for the original problem, not the
    simplified problem.
3. Strictly follow the input/output or function-signature contract.
4. Return only complete Python3 code. Do not add explanation or
    markdown fences.
```