

Controlled Self-Evolution for Algorithmic Code Optimization

Tu Hu^{1*}, Ronghao Chen^{2,8*}, Shuo Zhang^{9*}, Rui Xu⁴,
Jianghao Yin⁵, Mou Xiao Feng⁹, Jingping Liu⁶, Wenqi Jiang¹,
Shaolei Zhang⁷, Yuqi Fang^{1†}, Sen Hu^{3,8†}, Yi Xu^{9†}, Huacan Wang^{9†}

¹Nanjing University, ²International Academy of Phronesis Medicine (Guang Dong),
³Peking University, ⁴Fudan University, ⁵East China Normal University,
⁶Sun Yat-sen University, ⁷Renmin University of China, ⁸QuantaAlpha, ⁹Midea-AIRC

*These authors contributed equally to this work.

†Correspondence: yqfang@nju.edu.cn, husen@pku.edu.cn, xuyi42@midea.com, wanghc141@midea.com

Abstract

Self-evolution methods enhance code generation through iterative “generate-verify-refine” cycles, yet existing approaches suffer from low exploration efficiency, failing to discover algorithmically superior solutions within limited budgets. We identify the root cause as the lack of systematic controllability: execution feedback is not reliably converted into targeted search decisions across route planning, edit granularity, and experience reuse. We propose **Controlled Self-Evolution (CSE)**, a framework that couples three components into an integrated control architecture. Diversified Planning Initialization provides route diversity through structurally distinct algorithmic strategies. Modular Evolution provides feedback attribution and module-level editing by decomposing programs into functional modules and applying targeted mutation and compositional crossover. Hierarchical Evolution Memory maintains search state within tasks and distills reusable optimization patterns across tasks. Experiments on EffiBench-X demonstrate that CSE consistently outperforms all baselines across various LLM backbones. Further analysis reveals non-additive synergy among control layers, confirming that CSE’s effectiveness arises from component coupling rather than independent contributions. The code and evaluation implementation are publicly available at <https://github.com/QuantaAlpha/EvoControl>.

1 Introduction

Large Language Models (LLMs) (Comanici et al., 2025; Achiam et al., 2023; Cai et al., 2024; Grattafiori et al., 2024; Team et al., 2025; Guo et al., 2025a) have demonstrated strong code generation capabilities (Chen et al., 2021; Jiang et al., 2026a; Dong et al., 2025; Wang et al., 2025b; Hui et al., 2024; Wang et al., 2025a,c), yet single-turn generation (Luo et al., 2024; Wei et al., 2023) struggles with complex algorithmic problems and cannot

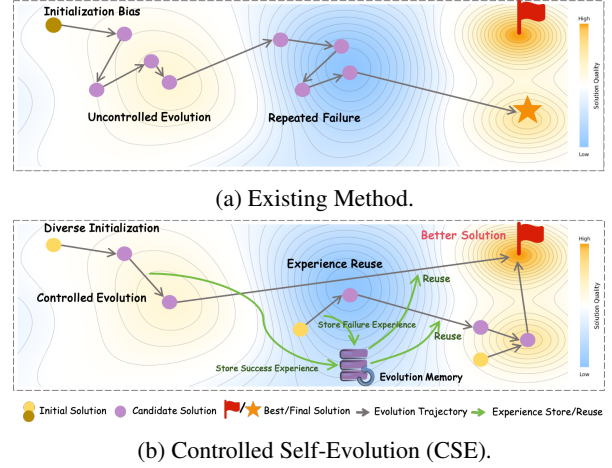


Figure 1: **Controlled Self-Evolution (CSE) improves exploration efficiency.** (a) Existing self-evolution suffers from Initialization Bias, Uncontrolled Evolution, and Repeated Failure, so candidates wander across the solution-quality landscape without converging. (b) CSE replaces these with Diversified Planning Initialization, Modular Evolution, and a Hierarchical Evolution Memory that stores success and failure experience for reuse, steering the trajectory toward a better solution within the same budget.

leverage execution feedback. Self-evolution methods (Gao et al., 2025; Fang et al., 2025), adopted by both code optimization agents (Du et al., 2025; Guo et al., 2025b; Novikov et al., 2025) and LLM-driven algorithm design (Romera-Paredes et al., 2023; Liu et al., 2024b; Ye et al., 2024; Stein and Bäck, 2025; Liu et al., 2026), address this through iterative “generate-verify-refine” cycles that cast code generation as feedback-driven search.

Yet this search process is computationally expensive, and practical deployment imposes strict resource constraints, demanding methods with higher exploration efficiency that discover better solutions earlier and keep improving within the budget.

Unfortunately, existing self-evolution methods suffer from critically low exploration efficiency un-

der such budgets (as shown in Figure 1). The root cause is not the absence of evolutionary primitives (diversity, selection, mutation, crossover, and reflective feedback have all been explored in prior work (Romera-Paredes et al., 2023; Liu et al., 2024b; Ye et al., 2024; Stein and Bäck, 2025)), but the **lack of systematic controllability** over how execution feedback is converted into future search actions. We observe that systematic controllability can be understood as progressively refining raw execution feedback into actionable search decisions through five synergistic levels: *route diversity* ensures structurally distinct solution regions are covered; *feedback attribution* localizes coarse signals to responsible functional components; *module-level targeted editing* translates diagnoses into precise mutations instead of whole-program rewrites; *search state maintenance* accumulates edit histories into navigational awareness of what has been tried and what to explore next; and *cross-task experience transfer* distills this knowledge into reusable optimization patterns. These levels exhibit bidirectional dependencies, where upstream levels provide the structured input that downstream levels require, while downstream levels enhance upstream decisions by supplying attributable structure and route-level awareness. This mutual reinforcement suggests that controllability is best addressed as an integrated architecture. However, existing methods address these levels in isolation or leave several entirely uncontrolled: diversity is introduced at initialization but not maintained; feedback drives whole-program rewriting without module-level attribution; and search state is not tracked, leading to redundant exploration.

We propose **Controlled Self-Evolution (CSE)**, a framework that couples three components into an integrated control loop to jointly achieve the synergy described above. Diversified Planning Initialization provides *route diversity* by generating structurally distinct algorithmic strategies, each serving as a trackable exploration direction maintained by memory throughout evolution. Modular Evolution provides *feedback attribution* and *module-level targeted editing*: it decomposes programs into functional modules, diagnoses bottleneck modules, and applies targeted mutation or compositional crossover under interface constraints. Hierarchical Evolution Memory provides *search state maintenance* and *cross-task experience transfer*: local memory tracks attempted directions and their outcomes; global memory distills cross-task patterns

for retrieval in new tasks. These components form a closed loop: planning provides trackable routes for memory; memory provides directional guidance for evolution; evolution provides module-attributed feedback for memory.

We conduct comprehensive experiments on EffiBench-X (QING et al., 2026), showing that CSE consistently outperforms all baselines across diverse LLM backbones. Further analysis confirms that CSE’s gains arise from the coupled control architecture: removing any component produces non-additive performance degradation.

2 Related Work

Code Generation with LLMs. Large Language Models have demonstrated remarkable capabilities in code generation (Dong et al., 2025; Liu et al., 2025; Lyu et al., 2025; Zhu et al., 2025; Lavon et al., 2025; Yao et al., 2025). Recent advances span instruction tuning (WizardCoder (Luo et al., 2024), Magicoder (Wei et al., 2023)), retrieval augmentation (Lu et al., 2022), while specialized models such as GPT-4o (Hurst et al., 2024) and DeepSeek-Coder (Guo et al., 2024) have achieved strong results on programming benchmarks. However, recent evaluations consistently reveal that LLMs tend to generate solutions that are "correct yet inefficient." For instance, EffiBench-X (QING et al., 2026) highlights a significant efficiency gap between model-generated code and human-written canonical solutions. These findings underscore that code generation capability does not equate to evolution proficiency, necessitating the introduction of post-generation systematic refinement mechanisms.

LLM-Driven Evolutionary Search. Self-evolution methods for code optimization (Gao et al., 2025; Fang et al., 2025) and LLM-driven automated algorithm design (Liu et al., 2026; Lehman et al., 2023) share a common paradigm: embedding LLMs in iterative “generate-evaluate-refine” loops to search over programs. At the iterative refinement level, self-reflection approaches (Madaan et al., 2023; Shinn et al., 2023; Sun et al., 2023) enable models to learn from execution feedback, while greedy methods such as AfterBurner (Du et al., 2025) and EffiLearner (Huang et al., 2024) perform local refinement on single candidates. To broaden exploration, population-based methods maintain candidate pools: FunSearch (Romera-Paredes et al., 2023) searches over programs with

automated evaluation, AEL (Liu et al., 2023) and EoH (Liu et al., 2024b) evolve heuristic ideas alongside code implementations, and AlphaEvolve (Novikov et al., 2025) and OpenEvolve (Sharma, 2025) demonstrate LLM-driven coding agents for algorithmic discovery. Diversity-driven strategies (Dat et al., 2025; Lange et al., 2026; Li et al., 2025), trajectory-level evolution (Guo et al., 2025b), and structured search via Monte Carlo tree search (Zheng et al., 2025b) further expand the search space. To improve variation operators, LLaMEA (Stein and Bäck, 2025) and LLaMEA-BO (Li et al., 2026) evolve complete algorithm implementations through LLM-driven mutation and crossover. To leverage accumulated feedback, ReEvo (Ye et al., 2024) augments evolutionary search with short-term and long-term verbal reflections, extended by metacognitive (Qiu et al., 2025), memetic-reflective (Qi et al., 2025), and hindsight-foresight prompting (ChentongChen et al., 2026) architectures, while co-evolutionary approaches (Huang et al., 2026; Chen et al., 2023; Yang et al., 2024) explore joint optimization of complementary artifacts. Each of these methods advances a specific aspect of the evolutionary process. CSE targets the integration challenge: rather than improving any single primitive, it couples route-level diversity, module-level feedback attribution, and hierarchical search state maintenance into a unified control architecture where each component’s effectiveness depends on the presence of the others.

3 Problem Formulation

We formalize the algorithmic code efficiency optimization task as follows. Given a problem specification x describing functional requirements and constraints, the goal is to generate an implementation y that satisfies correctness while achieving optimal execution efficiency.

The optimization environment is defined as $\mathcal{O} = (\mathcal{X}, \mathcal{Y}, \mathcal{F})$, where $\mathcal{X}$ represents the problem specification space, $\mathcal{Y}$ denotes the solution space, and $\mathcal{F} : \mathcal{Y} \times \mathcal{X} \rightarrow \mathbb{R}$ is a reward function evaluating solution quality. The reward function captures both correctness and efficiency.

The optimization unfolds as a population-based evolutionary process over T iterations. At each step t , we maintain a population $\mathcal{P}_t = \{y_1^{(t)}, y_2^{(t)}, \dots, y_{N_t}^{(t)}\}$ of N_t candidate solutions. The evolutionary trajectory $\mathcal{T} = (\mathcal{P}_0, \mathcal{P}_1, \dots, \mathcal{P}_T)$

progresses through feedback-driven selection, mutation, and crossover operators. Our objective is to discover the optimal solution $y^* = \arg \max_{y \in \bigcup_{t=0}^T \mathcal{P}_t} \mathcal{F}(y, x)$, requiring efficient navigation of the solution space through controlled rather than stochastic exploration.

4 Method

As illustrated in Figure 2, CSE progressively refines execution feedback into actionable search decisions through three tightly coupled components. Diversified Planning Initialization generates diverse initial solutions through varied planning strategies, providing broad coverage of the solution space. Modular Evolution uses reflection to decompose candidates into functional modules and applies targeted edits to the modules implicated by feedback, enabling precise and predictable optimization at each step. Hierarchical Evolution Memory maintains the search state within a task and distills reusable optimization patterns across tasks, providing directional guidance for the evolutionary process.

4.1 Diversified Planning Initialization

To maximize the diversity of initial solutions and ensure broad coverage of the solution space, we employ a two-stage approach: diverse planning followed by completion. Specifically, we prompt the agent $\mathcal{A}_\theta$ with explicit diversity constraints to generate a set of high-level solution sketches $\mathcal{Z} = \{z_1, \dots, z_{N_{\text{init}}}\}$, where each sketch represents a semantically distinct strategy.

Subsequently, each solution sketch is instantiated into a concrete code implementation, forming the initial candidate solutions:

$$y_i^{(0)} \sim \mathcal{A}_\theta(y \mid x, z_i), \quad \forall i \in \{1, \dots, N_{\text{init}}\}, \quad (1)$$

where $y_i^{(0)}$ denotes the complete output generated by the model conditioned on the input specification x and solution sketch z_i at the initial stage. Through this approach, the initial population $\mathcal{P}_0 = \{y_1^{(0)}, \dots, y_{N_{\text{init}}}^{(0)}\}$ achieves diverse coverage of the solution space. Each initial strategy serves as a trackable exploration direction, with its evolution trajectory maintained by local memory (§4.3).

4.2 Modular Evolution

Modular Evolution converts execution feedback into a functional diagnosis, decomposes the program into semantically meaningful modules, and

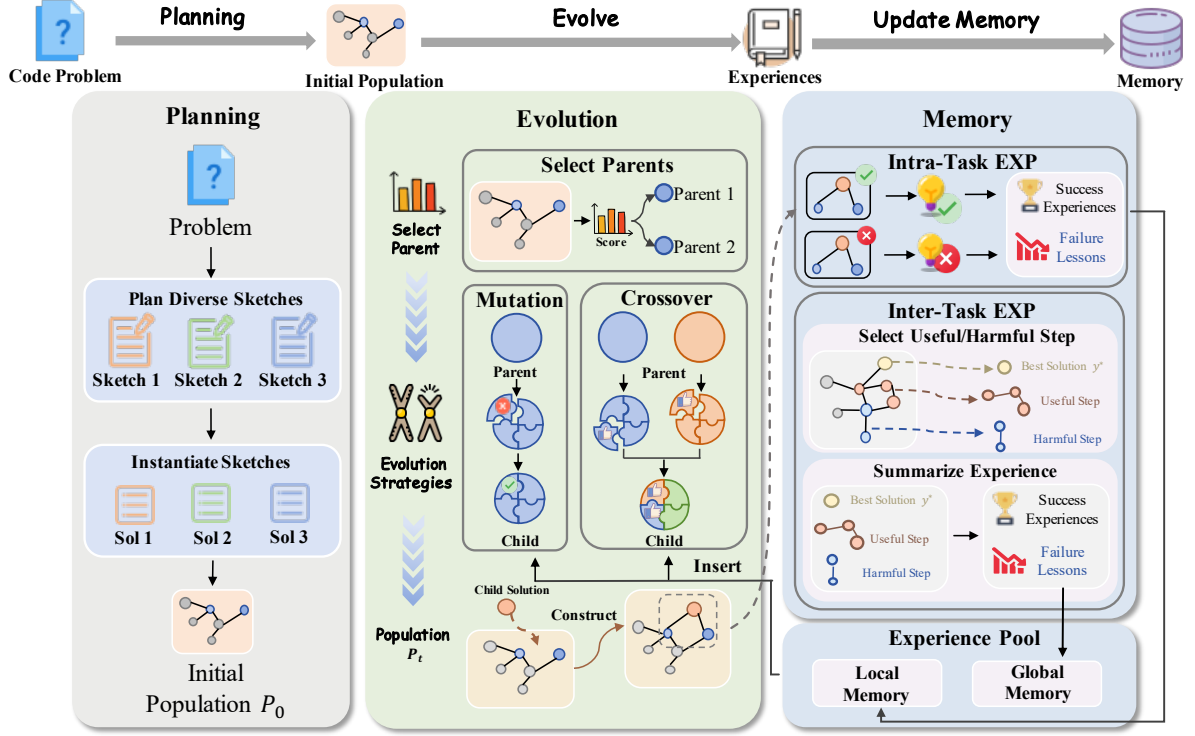


Figure 2: **Overview of CSE.** Three components form a control loop along the feedback refinement chain: Diversified Planning Initialization provides route diversity, Modular Evolution provides feedback attribution and module-level editing, and Hierarchical Evolution Memory maintains search state and enables cross-task experience transfer.

applies targeted edits to the modules implicated by the diagnosis, enabling precise and predictable optimization at each evolutionary step. At each step, one or two parents are sampled from the population with probability proportional to reward, then processed through the following pipeline.

Reflection-Guided Decomposition. Given a selected candidate y , CSE first asks the agent to analyze the current feedback, compare the candidate against the task constraints and available reference signals, and decompose the program into functional modules $\{c_1, c_2, \dots, c_m\}$. Each module is tagged with its role, interface assumptions, observed strengths, and suspected bottlenecks. This decomposition creates explicit edit boundaries that preserve working logic while focusing optimization pressure on the components most responsible for the current performance profile.

Module-Level Controlled Mutation. The agent uses the reflection diagnosis to identify the component c_{target} most likely responsible for low reward. CSE then regenerates only this target module while preserving all other modules and interfaces:

$$y_{\text{child}} = \{c_1, \dots, \text{Refine}(c_{\text{target}} \mid r_t), \dots, c_m\}, \quad (2)$$

where r_t denotes the reflection-derived rationale and feedback evidence at step t . This module-level constraint ensures that the model explicitly identifies which component is being changed, justifies why the feedback implicates it, and verifies that the revised module remains compatible with the preserved code.

Module-Level Compositional Crossover. To facilitate the flow of advantageous traits across the population, CSE performs module-level recombination. When parent solutions y_A and y_B exhibit complementary advantages in different components, the crossover operator aligns the decomposed modules by function, then synthesizes a child from selected compatible modules:

$$y_{\text{child}} = \text{Compose}(\{c_{\text{time}}^{(A)}\}, \{c_{\text{robust}}^{(B)}\} \mid r_t), \quad (3)$$

where $c_{\text{time}}^{(A)}$ denotes the time-efficient component from solution y_A , and $c_{\text{robust}}^{(B)}$ represents the robustness-oriented component from solution y_B . The reflection rationale r_t constrains the composition to preserve interfaces and resolve conflicts between the imported modules.

4.3 Hierarchical Evolution Memory

To maintain search state within a task and distill reusable optimization patterns across tasks, we employ Hierarchical Evolution Memory.

Local Memory. $\mathcal{M}_{\text{local}}$ maintains the search state of the current task by recording what has been tried and what each attempt revealed. At each evolutionary step t , we compare the reward change $\Delta_t = \mathcal{F}(y_{\text{child}}^{(t)}, x) - \mathcal{F}(y_{\text{parent}}^{(t)}, x)$ between the parent $y_{\text{parent}}^{(t)}$ and its child $y_{\text{child}}^{(t)}$. The agent distills each comparison into a structured experience entry containing two types of insights: success insights ($\Delta_t > 0$) that capture what led to improvements, and failure lessons ($\Delta_t \leq 0$) that capture why a direction degraded performance. Together, these entries form a cumulative record of the exploration trajectory. This reflection process is formalized as:

$$m_t \leftarrow \text{Reflect}(y_{\text{child}}^{(t)}, y_{\text{parent}}^{(t)}, \Delta_t), \quad (4a)$$

$$\mathcal{M}_t^{\text{loc}} \leftarrow \text{Compress}(\mathcal{M}_{t-1}^{\text{loc}} \cup \{m_t\}), \quad (4b)$$

where m_t denotes the distilled experience at step t . At evolution step $t + 1$, the accumulated $\mathcal{M}_{\text{local}}$ is dynamically injected into the prompt context, providing directional guidance: success insights direct the model toward validated optimization strategies, while failure lessons steer it away from previously explored dead ends. To prevent memory overflow as iterations accumulate, whenever $\mathcal{M}_t^{\text{loc}}$ exceeds a predefined length threshold, we perform semantic compression to keep local memory dense.

Global Memory. $\mathcal{M}_{\text{glb}}$ aims to distill and reuse inter-task experiences. For each task τ , we collect all evolution steps and their reward changes. We keep the top- K improving useful steps and top- K degrading harmful steps (ranked by Δ), denoted by $\mathcal{S}_{\tau}^{+}$ and $\mathcal{S}_{\tau}^{-}$, respectively. The LLM distills a task-level global experience g_{τ} with $\mathcal{S}_{\tau}^{+}$, $\mathcal{S}_{\tau}^{-}$ and best solution y_{τ}^{*} and stores it in a vector database:

$$g_{\tau} \leftarrow \text{Reflect}(\mathcal{S}_{\tau}^{+}, \mathcal{S}_{\tau}^{-}, y_{\tau}^{*}), \quad (5a)$$

$$\mathcal{M}^{\text{glb}} \leftarrow \mathcal{M}^{\text{glb}} \cup \{g_{\tau}\}. \quad (5b)$$

At evolutionary step t , the agent generates N_q targeted retrieval queries based on the current evolutionary context (including the current code state, encountered errors, or performance bottlenecks):

$$\{q_t^{(n)}\}_{n=1}^{N_q} \leftarrow \text{GenerateQueries}(\text{Context}_t), \quad (6a)$$

$$\mathcal{E}_t^{\text{ret}} \leftarrow \text{Retrieve}(\mathcal{M}^{\text{glb}}, \{q_t^{(n)}\}_{n=1}^{N_q}), \quad (6b)$$

where $\{q_t^{(n)}\}_{n=1}^{N_q}$ represents the generated queries and $\mathcal{E}_t^{\text{ret}}$ denotes the retrieved relevant experiences. This mechanism ensures that the agent can precisely invoke past experiences from similar tasks when encountering specific challenges. Appendix C provides the implementation details of global memory extraction, query generation, retrieval, and context injection.

The full end-to-end procedure is provided in Appendix A and Algorithm 1; the main text focuses on the three control mechanisms that turn execution feedback into route-level diversity, module-level edits, and accumulated search guidance.

5 Experiments

5.1 Experimental Setup

Benchmark. We evaluate on EffiBench-X (QING et al., 2026), which aggregates 623 algorithmic problems from major competitive programming platforms, including AtCoder, Codeforces, and LeetCode, spanning six programming languages. Each problem comes with strict time/memory limits and comprehensive test cases. We run experiments on two different languages (Python and C++) to assess the generality of our method. Following prior work (QING et al., 2026), we report three normalized efficiency metrics: Execution-Time ratio (ET), Memory-Peak ratio (MP) and Memory-Integral ratio (MI), which compare the LLM-generated solution with the human reference on every problem. Detailed definitions and derivations are provided in Appendix B.

Baselines. We compare CSE against three code-generation paradigms: (1) Direct: a single-turn generation. (2) Self-Reflection: iterative modification of the current best solution based on its feedback (e.g., variants of EffiLearner (Huang et al., 2024) and AfterBurner (Du et al., 2025)). (3) SE-Agent (Guo et al., 2025b): a trajectory-level self-evolving agent method (4) AlphaEvolve (Novikov et al., 2025): a representative self-evolving agent method; we reproduce it using the open-source implementation of OpenEvolve (Sharma, 2025). We evaluate two open-source models (DeepSeek-V3-0324 (Liu et al., 2024a) and Qwen3-235B-A22B (Yang et al., 2025)) and two closed-source models (Claude-4.5-Sonnet and GPT-5).

Implementation Details. All methods (except Direct) share a 30-candidate budget per task; the best solution encountered in the trajectory is

	Python				C++				Avg.			
	ET	MP	MI	Tok.	ET	MP	MI	Tok.	ET	MP	MI	Tok.
Qwen3-235B-A22B												
Direct	31.10	49.13	46.28	–	17.42	46.55	19.24	–	24.26	47.84	32.76	–
Self-Reflection	48.49	49.82	50.89	47.2K	47.47	48.00	44.75	48.3K	47.98	48.91	47.82	47.8K
SE-Agent	48.57	<u>50.39</u>	49.63	137.5K	50.71	48.25	<u>47.19</u>	128.2K	49.64	49.32	48.41	132.9K
AlphaEvolve	<u>48.70</u>	50.33	51.47	109.1K	<u>48.75</u>	<u>48.67</u>	46.58	114.0K	48.73	49.50	49.03	111.6K
CSE (Ours)	49.06	50.99	52.38	137.3K	48.70	49.03	47.95	125.4K	<u>48.88</u>	50.01	50.17	131.4K
DeepSeek-v3-0324												
Direct	52.90	58.89	56.22	–	44.21	45.79	41.01	–	48.56	52.34	48.62	–
Self-Reflection	57.35	<u>60.20</u>	<u>61.18</u>	56.0K	45.43	<u>46.53</u>	43.16	54.0K	51.39	<u>53.37</u>	52.17	55.0K
SE-Agent	<u>60.09</u>	59.68	<u>60.69</u>	118.4K	<u>47.74</u>	46.42	<u>44.54</u>	117.9K	<u>53.92</u>	<u>53.05</u>	<u>52.62</u>	118.2K
AlphaEvolve	58.86	60.04	60.24	106.3K	46.19	46.41	44.02	107.2K	52.53	53.23	52.13	106.8K
CSE (Ours)	61.06	61.03	62.80	115.3K	48.75	46.89	47.37	120.9K	54.91	53.96	55.09	118.1K
Claude-4.5-Sonnet												
Direct	65.43	69.03	67.31	–	63.16	<u>69.16</u>	62.78	–	64.30	69.10	65.05	–
Self-Reflection	<u>69.79</u>	71.16	<u>73.08</u>	65.0K	70.29	<u>68.42</u>	67.43	71.6K	70.04	<u>69.79</u>	<u>70.26</u>	68.3K
SE-Agent	<u>64.14</u>	<u>70.58</u>	67.25	116.0K	<u>72.93</u>	68.81	<u>70.61</u>	119.2K	68.54	<u>69.70</u>	<u>68.93</u>	117.6K
AlphaEvolve	69.45	69.50	70.57	124.3K	71.07	68.45	67.66	116.9K	<u>70.26</u>	68.98	69.12	120.6K
CSE (Ours)	71.17	70.39	73.94	139.4K	75.29	69.82	74.88	130.5K	73.23	70.11	74.41	135.0K
GPT-5												
Direct	60.21	62.31	61.66	–	60.02	62.60	56.60	–	60.12	62.46	59.13	–
Self-Reflection	62.21	63.09	63.72	45.8K	61.79	62.08	56.76	44.2K	62.00	62.59	60.24	45.0K
SE-Agent	61.74	<u>64.51</u>	65.36	129.0K	<u>68.61</u>	62.70	<u>63.46</u>	133.6K	<u>65.18</u>	<u>63.61</u>	<u>64.41</u>	131.3K
AlphaEvolve	<u>63.30</u>	63.96	<u>65.53</u>	114.5K	64.13	<u>63.05</u>	<u>60.54</u>	114.1K	63.72	63.51	63.04	114.3K
CSE (Ours)	65.46	66.82	68.10	137.8K	69.94	64.42	66.83	138.2K	67.70	65.62	67.47	138.0K

Table 1: **Main results on EffiBench-X (Python and C++)**. ET, MP, and MI measure execution time, peak memory, and memory integral ratio relative to human solutions. Tok. is the average total tokens per task. Avg. reports the per-column mean across Python and C++. Best results are in bold and second-best results are underlined.

recorded. Following EffiLearner’s setup (Huang et al., 2024), efficiency is measured only on tasks already solved by Direct to decouple speed from correctness; if a method exhausts its budget without a valid solution, we fall back to the Direct baseline for that task, guaranteeing identical problem sets across evaluators. Across all methods, the reward is the raw memory–time integral, inverted into a maximization score to capture joint runtime and memory gains. See Appendix C for more details.

5.2 Main Results

Table 1 presents the main experimental results on EffiBench-X across Python and C++ under a fixed test-time budget of 30 candidates per task. CSE consistently achieves the best performance across most efficiency metrics (ET, MP, MI) on both programming languages, with particularly notable improvements on memory integral, demonstrating its effectiveness in discovering algorithmically efficient solutions within limited exploration budgets. The advantages hold across diverse model capabilities, from open-source mod-

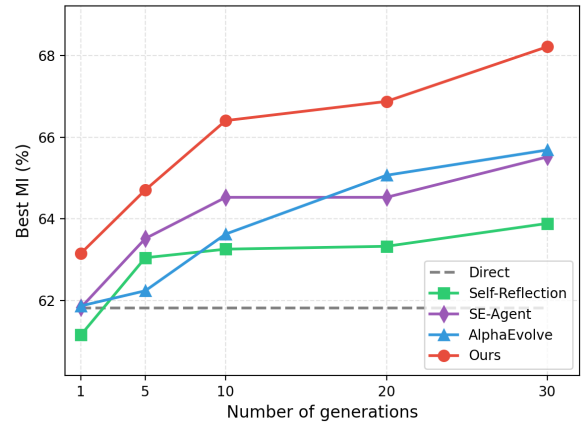


Figure 3: **Best-so-far MI vs. generations**. At each generation t , we report the *best-so-far* MI.

els (Qwen3-235B-A22B, DeepSeek-v3-0324) to state-of-the-art closed-source models (Claude-4.5-Sonnet, GPT-5), validating that CSE’s control architecture is model-agnostic. The gains are most pronounced with stronger backbones (e.g., +12.10 MI on Claude-4.5-Sonnet C++), suggesting that more

Method	LCB v5	LCB v6
Direct	76/167 (45.51%)	78/175 (44.57%)
Self-Reflection	97/167 (58.08%)	97/175 (55.43%)
SE-Agent	109/167 (65.27%)	108/175 (61.71%)
AlphaEvolve	104/167 (62.28%)	98/175 (56.00%)
CSE (Ours)	115/167 (68.86%)	109/175 (62.29%)

Table 2: **LiveCodeBench code-generation results.**

capable models benefit disproportionately from systematic search control that channels generation into targeted, feedback-grounded edits.

Beyond the final metrics, Figure 3 reveals CSE’s convergence advantage: CSE surpasses the generation-30 performance of all competing methods by generation 20 and continues to improve where others plateau. This sustained improvement reflects the interplay of route-level diversity and search state maintenance: diverse initial strategies prevent early convergence to a single basin, while accumulated local memory steers subsequent iterations away from explored directions and toward unexploited opportunities. CSE’s per-task token consumption is comparable to other population-based methods (e.g., on GPT-5: CSE 138.0K vs. SE-Agent 131.3K vs. AlphaEvolve 114.3K). The additional tokens are invested in maintaining search diversity and enriching each evolutionary step with higher information density. Importantly, if additional tokens alone explained the improvement, CSE would not surpass competitors’ 30-generation results with only two-thirds of the iteration budget, confirming that the gains are attributable to the control architecture. We examine repeated-run stability in Appendix E, confirming that the observed gains are not dominated by stochastic variation.

5.3 Further Analysis

Beyond Efficiency: Correctness-Oriented Evaluation. To examine whether CSE extends to correctness-oriented code optimization, we evaluate DeepSeek-V3-0324 on LiveCodeBench (Jain et al., 2025) release v5 and v6 under the same 30-candidate budget. As shown in Table 2, CSE achieves the highest pass rate on both releases. This result suggests that CSE is not limited to efficiency optimization on EffiBench-X, but also extends to broader code optimization scenarios where functional correctness is the primary objective. Appendix G further reports an AIME 2024 extension, testing the same controlled evolution idea in mathematical reasoning with verifiable answer checking.

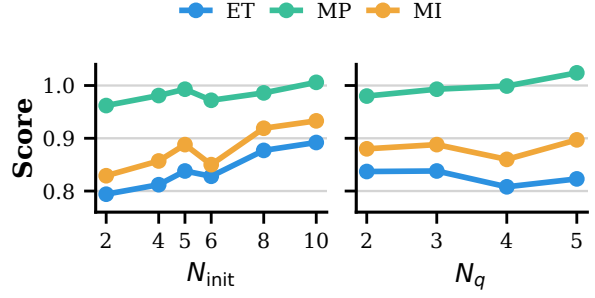


Figure 4: **Hyperparameter sensitivity of CSE.**

Setting	ET	MP	MI
CSE (Full)	65.46%	66.82%	68.10%
w/o <i>Planning</i>	61.39%	64.40%	63.82%
w/o <i>Evolution</i>	62.16%	64.28%	64.72%
w/o <i>Memory</i>	59.87%	64.75%	63.08%

Table 3: **Ablation study.** *Planning* means diversified planning initialization; *Evolution* means modular evolution; *Memory* means hierarchical evolution memory.

Hyperparameter Sensitivity. Figure 4 varies two search-control hyperparameters: the number of diversified initial candidates N_{init} and the number of memory-retrieval queries. Across configurations, CSE maintains strong performance, and the overall trend does not depend on a narrowly tuned setting. Increasing N_{init} tends to improve performance, which is expected because a larger initial pool provides broader coverage of candidate solution strategies. In contrast, increasing the query count yields a smaller and less monotonic effect, suggesting that retrieval helps but is not the dominant source of gains. Importantly, our default setting is not the best-performing configuration in this sweep; therefore, the main results are not obtained by selecting the optimal hyperparameters on the evaluation set. Overall, CSE shows only moderate hyperparameter sensitivity, supporting robustness across reasonable search-control settings.

Ablation Studies on CSE. To isolate each component’s role, we remove one at a time under the same 30-candidate budget (Table 3). Each removal disrupts a distinct level of the information refinement chain. Removing *Planning* eliminates route diversity, confining search to a narrow region of the solution space (−4.28 MI). Removing *Evolution* disables feedback attribution and module-level editing, forcing the system to fall back on whole-program rewriting (−3.38 MI). Removing *Memory* eliminates search state maintenance, causing the system to revisit previously explored directions

Method	#Imp.	Iter@Best	Last-10 #Imp.
SE-Agent	1.60	9.51	0.19
AlphaEvolve	0.90	7.44	0.06
CSE (Ours)	1.79	12.06	0.29

Table 4: **Statistics of Evolution dynamics.**

without awareness of what has already been tried (-5.02 MI). The consistent degradation across both ET and MP confirms that these are not independent add-ons but interlocking control layers: each transforms feedback at a different granularity, and removing any one breaks the refinement chain. We additionally provide qualitative case studies in Appendix D to visualize the evolution dynamics under each ablation. Appendix N further presents a concrete CSE evolution trajectory.

Non-Additive Synergy among Control Layers.

Our central claim is that CSE’s components form a coupled control architecture rather than independent add-ons. To test this, we compare different component combinations under the same 30-iteration budget. Figure 5 reports MI and the gain from adding *Memory* (Δ MI) to each combination. The results reveal a clear non-additive pattern. Adding *Memory* to *Planning* alone provides negligible benefit (Δ MI = -0.23): without module-level feedback attribution, retrieved experiences cannot be grounded into concrete edit operations. Combining *Memory* with *Evolution* yields substantial gains (Δ MI = $+2.68$): feedback attribution and module-level editing provide the interface through which accumulated search state can direct precise modifications. The strongest effect emerges when all three components are present (Δ MI = $+5.02$), where route diversity ensures that memory entries cover structurally distinct strategies and module-level evolution provides the actionable interface to apply retrieved experience. This super-additive pattern provides direct evidence for bidirectional dependencies among control layers. We additionally isolate the cross-task component in Appendix F, where a focused ablation evaluates the effect of injecting retrieved global memory.

Exploration on Evolution Dynamics. To understand when and how frequently improvements occur during evolution, we conduct a fine-grained analysis over 30-iteration runs. Table 4 reports three statistics: (i) the number of improvement times (#Imp.), measuring how many iterations yield efficiency gains over the previous best; (ii) the itera-



Figure 5: **Non-additive synergy among control layers.** We report MI and the gain Δ MI from adding *Memory* to each component combination.

tion at which the final solution is found (Iter@Best), indicating whether the best result appears early or late; and (iii) improvements in the last 10 generations (Last-10 #Imp.), quantifying the ability to sustain late-stage progress. The results show that CSE achieves substantially more frequent improvements than baselines (#Imp. = 1.79 vs. 0.90 for AlphaEvolve and 1.60 for SE-Agent) and notably stronger late-stage progress (Last-10 #Imp. = 0.29 vs. 0.06 for AlphaEvolve and 0.19 for SE-Agent). This sustained late-stage progress can be attributed to search state maintenance: as local memory accumulates success and failure records, later iterations operate with richer directional guidance, enabling the system to continue identifying productive edits even as early opportunities are exhausted.

6 Conclusion

We present Controlled Self-Evolution (CSE), a framework that addresses the low exploration efficiency of existing self-evolution methods by introducing systematic controllability over how execution feedback is converted into search decisions. CSE couples three components into an integrated control architecture: Diversified Planning Initialization provides route diversity, Modular Evolution provides feedback attribution and module-level editing, and Hierarchical Evolution Memory maintains search state and enables cross-task experience transfer. These components form a closed loop where each level’s effectiveness depends on the others, as confirmed by the non-additive synergy in our analysis. Experiments on EffiBench-X demonstrate that CSE consistently outperforms all baselines across various LLM backbones.

Limitations

CSE can continuously improve solution quality through multi-round evolution. However, we have not yet explored how to amortize this iterative optimization into the base model. A promising yet underexplored direction is to distill CSE’s evolution trajectories into RL-style training (Yu et al., 2025; Zheng et al., 2025a; Jiang et al., 2026b) signals to strengthen the base model, enabling comparable or better optimization and producing higher-quality solutions.

Another limitation concerns responsible deployment. Automated code optimization can produce candidates that pass available tests while relying on brittle assumptions, hidden input-distribution biases, or implementation choices that are difficult for downstream users to audit. Such failures may harm users who deploy optimized code in safety-critical or resource-constrained settings. CSE mitigates this risk by keeping execution feedback in the loop, enforcing interface-preserving modular edits, and selecting candidates only after validation, but these safeguards do not replace human review, broader testing, or domain-specific safety checks before deployment.

References

- Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.
- Zheng Cai, Maosong Cao, Haojiong Chen, Kai Chen, Keyu Chen, Xin Chen, Xun Chen, Zehui Chen, Zhi Chen, Pei Chu, et al. 2024. Internlm2 technical report. *arXiv preprint arXiv:2403.17297*.
- Angelica Chen, David Dohan, and David So. 2023. *Evo-prompting: Language models for code-level neural architecture search*. In *Thirty-seventh Conference on Neural Information Processing Systems*.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.
- Chentong Chen, Mengyuan Zhong, Jialong Shi, Jianyong Sun, and Ye Fan. 2026. *Hifo-prompt: Prompting with hindsight and foresight for LLM-based automatic heuristic design*. In *The Fourteenth International Conference on Learning Representations*.
- Gheorghe Comanici, Eric Bieber, Mike Schaeckermann, Ice Pasupat, Naveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*.
- Pham Vu Tuan Dat, Long Doan, and Huynh Thi Thanh Binh. 2025. Hsevo: Elevating automatic heuristic design with diversity-driven harmony search and genetic algorithm using llms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 26931–26938. <https://github.com/datphamvn/HSEvo>.
- Yihong Dong, Xue Jiang, Jiaru Qian, Tian Wang, Kechi Zhang, Zhi Jin, and Ge Li. 2025. A survey on code generation with llm-based agents. *arXiv preprint arXiv:2508.00083*.
- Mingzhe Du, Anh Tuan Luu, Yue Liu, Yuhao QING, Dong HUANG, Xinyi He, Qian Liu, Zejun MA, and See-Kiong Ng. 2025. *Afterburner: Reinforcement learning facilitates self-improving code efficiency optimization*. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.
- Jinyuan Fang, Yanwen Peng, Xi Zhang, Yingxu Wang, Xinhao Yi, Guibin Zhang, Yi Xu, Bin Wu, Siwei Liu, Zihao Li, et al. 2025. A comprehensive survey of self-evolving ai agents: A new paradigm bridging foundation models and lifelong agentic systems. *arXiv preprint arXiv:2508.07407*.
- Huan-ang Gao, Jiayi Geng, Wenye Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Yiran Wu, et al. 2025. A survey of self-evolving agents: What, when, how, and where to evolve on the path to artificial super intelligence. *arXiv preprint arXiv:2507.21046*.
- Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.
- Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. 2025a. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.
- Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yifan Wu, YK Li, et al. 2024. Deepseek-coder: when the large language model meets programming—the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.
- Yifu Guo, Jiaye Lin, Huacan Wang, Yuzhen Han, Sen Hu, Ziyi Ni, Licheng Wang, and Mingguang Chen. 2025b. *SE-agent: Self-evolution trajectory optimization in multi-step reasoning with LLM-based agents*. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

- Dong Huang, Jianbo Dai, Han Weng, Puzhen Wu, Yuhao Qing, Heming Cui, Zhijiang Guo, and Jie M Zhang. 2024. Effilearner: Enhancing efficiency of generated code via self-optimization. *Advances in Neural Information Processing Systems*, 37:84482–84522.
- Ziyao Huang, Weiwei Wu, Kui Wu, Wei-Bin Lee, and Jianping Wang. 2026. [CALM: Co-evolution of algorithms and language model for automatic heuristic design](#). In *The Fourteenth International Conference on Learning Representations*.
- Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.
- Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.
- Naman Jain, Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. [Livecodebench: Holistic and contamination free evaluation of large language models for code](#). In *International Conference on Learning Representations*, volume 2025, pages 58791–58831.
- Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2026a. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*, 35(2):1–72.
- Xue Jiang, Yihong Dong, Mengyang Liu, Deng Hongyi, Tian Wang, Yongding Tao, Zhi Jin, Wenpin Jiao, and Ge Li. 2026b. Coderl+: Improving code generation via reinforcement with execution semantics alignment. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3608–3622.
- Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. 2026. [Shinkaevolve: Towards open-ended and sample-efficient program evolution](#). In *The Fourteenth International Conference on Learning Representations*.
- Boaz Lavon, Shahar Katz, and Lior Wolf. 2025. [Execution guided line-by-line code generation](#). In *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pages 147615–147640. Curran Associates, Inc.
- Joel Lehman, Jonathan Gordon, Shawn Jain, Kamal Ndousse, Cathy Yeh, and Kenneth O Stanley. 2023. Evolution through large models. In *Handbook of evolutionary machine learning*, pages 331–366. Springer.
- Kefan Li, Yuan Yuan, Hongyue Yu, Tingyu Guo, and Shijie Cao. 2025. Cocoevo: Co-evolution of programs and test cases to enhance code generation. *IEEE Transactions on Evolutionary Computation*.
- Wenhu Li, Niki van Stein, Thomas Bäck, and Elena Raponi. 2026. [Llamea-bo: A large language model evolutionary algorithm for automatically generating bayesian optimization algorithms](#). In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '26*, page 909–918, New York, NY, USA. Association for Computing Machinery.
- Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024a. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.
- Fei Liu, Xialiang Tong, Mingxuan Yuan, and Qingfu Zhang. 2023. Algorithm evolution using large language model. *arXiv preprint arXiv:2311.15249*.
- Fei Liu, Tong Xialiang, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. 2024b. [Evolution of heuristics: Towards efficient automatic algorithm design using large language model](#). In *Forty-first International Conference on Machine Learning*.
- Fei Liu, Yiming Yao, Ping Guo, Zhiyuan Yang, Zhe Zhao, Xi Lin, Xialiang Tong, Kun Mao, Zhichao Lu, Zhenkun Wang, Mingxuan Yuan, and Qingfu Zhang. 2026. [A systematic survey on large language models for algorithm design](#).
- Jiawei Liu, Nirav Diwan, Zhe Wang, Haoyu Zhai, Xiaona Zhou, Kiet A. Nguyen, Tianjiao Yu, Muntasir Wahed, Yinlin Deng, hadjer Benkraouda, Yuxiang Wei, LINGMING ZHANG, Ismini Lourentzou, and Gang Wang. 2025. [Purpcode: Reasoning for safer code generation](#). In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.
- Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seungwon Hwang, and Alexey Svyatkovskiy. 2022. Reacc: A retrieval-augmented code completion framework. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6227–6240.
- Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2024. Wizardcoder: Empowering code large language models with evolve-instruct. In *International Conference on Learning Representations*, volume 2024, pages 27168–27188.
- Zhiyi Lyu, Jianguo Huang, Yanchen Deng, Steven Hoi, and Bo An. 2025. [Let’s revise step-by-step: A unified local search framework for code generation with LLMs](#). In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.
- Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegrefe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2023. Self-refine: Iterative refinement with self-feedback. *Advances in neural information processing systems*, 36:46534–46594.

- Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. 2025. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*.
- Fubo Qi, Tianyu Wang, Ruixiang Zheng, and Mian Li. 2025. A memetic and reflective evolution framework for automatic heuristic design using large language models. *Applied Sciences*, 15(15):8735.
- Yuhao QING, Boyu Zhu, Mingzhe Du, Zhijiang Guo, Terry Yue Zhuo, Qianru Zhang, Jie M. Zhang, Heming Cui, Siu Ming Yiu, Dong HUANG, See-Kiong Ng, and Anh Tuan Luu. 2026. [Effibench-x: A multi-language benchmark for measuring efficiency of LLM-generated code](#). In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.
- Zishang Qiu, Xinan Chen, Long Chen, and Ruibin Bai. 2025. Mela: A metacognitive llm-driven architecture for automatic heuristic design. *arXiv preprint arXiv:2507.20541*.
- Bernardino Romera-Paredes, Mohammadamin Barekatin, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. 2023. [Mathematical discoveries from program search with large language models](#). *Nature*.
- Asankhaya Sharma. 2025. [Openevolve: an open-source evolutionary coding agent](#).
- 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.
- Niki van Stein and Thomas Bäck. 2025. [Llamea: A large language model evolutionary algorithm for automatically generating metaheuristics](#). *IEEE Transactions on Evolutionary Computation*, 29(2):331–345.
- Haotian Sun, Yuchen Zhuang, Ling kai Kong, Bo Dai, and Chao Zhang. 2023. Adaplaner: Adaptive planning from feedback with language models. *Advances in neural information processing systems*, 36:58202–58245.
- Kimi Team, Angang Du, Bofei Gao, Bowei Xing, Changjiu Jiang, Cheng Chen, Cheng Li, Chenjun Xiao, Chenzhuang Du, Chonghua Liao, et al. 2025. Kimi k1. 5: Scaling reinforcement learning with llms. *arXiv preprint arXiv:2501.12599*.
- Evan Wang, Federico Cassano, Catherine Wu, Yunfeng Bai, William Song, Vaskar Nath, Ziwen Han, Sean Hendryx, Summer Yue, and Hugh Zhang. 2025a. Planning in natural language improves llm search for code generation. In *International Conference on Learning Representations*, volume 2025, pages 2432–2478.
- Huanting Wang, Jingzhi Gong, Huawei Zhang, Jie Xu, and Zheng Wang. 2025b. Ai agentic programming: A survey of techniques, challenges, and opportunities. *arXiv preprint arXiv:2508.11126*.
- Zhengren Wang, Rui ling, Chufan Wang, Yongan Yu, Sizhe Wang, Zhiyu li, Feiyu Xiong, and Wentao Zhang. 2025c. [Maintaincoder: Maintainable code generation under dynamic requirements](#). In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.
- Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Empowering code generation with oss-instruct. *arXiv preprint arXiv:2312.02120*.
- An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.
- Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. 2024. [Large language models as optimizers](#). In *The Twelfth International Conference on Learning Representations*.
- Feng Yao, Zilong Wang, Liyuan Liu, Junxia Cui, Li Zhong, Xiaohan Fu, Haohui Mai, Viswanathan Krishnan, Jianfeng Gao, and Jingbo Shang. 2025. [Training language models to generate quality code with program analysis feedback](#). In *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pages 136036–136056. Curran Associates, Inc.
- Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. 2024. Reevo: Large language models as hyper-heuristics with reflective evolution. volume 37, pages 43571–43608.
- Qiyang Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gao-hong Liu, Juncai Liu, LingJun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, et al. 2025. [DAPO: An open-source LLM reinforcement learning system at scale](#). In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.
- Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong Liu, Rui Men, An Yang, Jingren Zhou, and Junyang Lin. 2025a. [Group sequence policy optimization](#). *Preprint*, arXiv:2507.18071.
- Zhi Zheng, Zhuoliang Xie, Zhenkun Wang, and Bryan Hooi. 2025b. [Monte Carlo tree search for comprehensive exploration in LLM-based automatic heuristic design](#). In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 78338–78373. PMLR.

Derui Zhu, Dingfan Chen, jinfu chen, Jens Grossklags, Walter Pretschner, and Weiyi Shang. 2025. [More than just functional: LLM-as-a-critique for efficient code generation](#). In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

A Algorithmic Overview of CSE

We summarize the end-to-end workflow of Controlled Self-Evolution (CSE) in Algorithm 1. CSE proceeds in three stages: (i) diversified planning initialization constructs a diverse initial population $\mathcal{P}_0$ of five candidates; (ii) 25 evolution steps perform reward-based parent selection, retrieve inter-task experiences from $\mathcal{M}^{\text{glb}}$ and inject them with $\mathcal{M}_{t-1}^{\text{loc}}$, then apply modular refinement (MODULARREFINE) or composition (MODULARCOMPOSE); and (iii) after the shared 30-candidate budget is exhausted, we return y^* and distill task-level experience g_τ from top- K improving/degrading steps to update $\mathcal{M}^{\text{glb}}$.

B Evaluation Metrics.

To quantify the efficiency of LLM-generated solutions relative to human-expert-written reference solutions, we follow prior work and report three normalized metrics: Execution Time (ET), Memory Peak (MP), and Memory Integral (MI). We evaluate on N problems, indexed by $i \in \{1, \dots, N\}$. For each problem i , we measure the execution time, peak memory usage, and memory-time integral for both the human reference solution (superscript H) and the LLM-generated solution (superscript L), under the same evaluation protocol.

Failure Handling. If the LLM-generated solution for problem i fails to pass all test cases or encounters a runtime error (e.g., timeout, crash), we assign a score of 0 for that problem for all metrics.

Clipping. To both (i) preserve the ability to credit solutions that outperform the human reference (ratios > 1) and (ii) limit the influence of extreme outliers, we apply

$$\text{clip}(z, 0, k) = \min(\max(z, 0), k).$$

In all experiments we set $k = 5$ (shared across all methods), which retains sufficient headroom for surpassing human references while preventing a small number of extreme cases from dominating the average.

(1) Execution Time (ET). Let T_i^H denote the raw execution time of the human reference solution required to pass all test cases for problem i , and T_i^L denote the execution time of the LLM-generated solution (conditioned on passing). The per-problem

ET score is

$$s_i^T = \begin{cases} 0, & \text{if solution fails,} \\ \text{clip}\left(\frac{T_i^H}{T_i^L}, 0, 5\right), & \text{otherwise,} \end{cases} \quad (7)$$

and the overall ET is the mean score across problems, reported in a scaled form for readability:

$$\text{ET}(\%) = \left(\frac{1}{N} \sum_{i=1}^N s_i^T \right) \times 100\%. \quad (8)$$

(2) Memory Peak (MP). Let M_i^H and M_i^L denote the raw peak memory usage of the human reference and the LLM-generated solution on problem i , respectively. The per-problem MP score is

$$s_i^M = \begin{cases} 0, & \text{if solution fails,} \\ \text{clip}\left(\frac{M_i^H}{M_i^L}, 0, 5\right), & \text{otherwise,} \end{cases} \quad (9)$$

and the overall MP is

$$\text{MP}(\%) = \left(\frac{1}{N} \sum_{i=1}^N s_i^M \right) \times 100\%. \quad (10)$$

(3) Memory Integral (MI). To measure lifetime memory consumption, we define the raw memory-time integral for one execution as

$$A = \int_0^{T_{\text{total}}} m(t) dt, \quad (11)$$

where $m(t)$ is the memory usage trace at time t and T_{total} is the total execution time. This integral is numerically approximated from high-resolution profiling traces. Let A_i^H and A_i^L denote the memory integrals of the human reference and the LLM-generated solution on problem i , respectively. The per-problem MI score is

$$s_i^A = \begin{cases} 0, & \text{if solution fails,} \\ \text{clip}\left(\frac{A_i^H}{A_i^L}, 0, 5\right), & \text{otherwise,} \end{cases} \quad (12)$$

and the overall MI is

$$\text{MI}(\%) = \left(\frac{1}{N} \sum_{i=1}^N s_i^A \right) \times 100\%. \quad (13)$$

C Implementation Details

Experimental Setup Rationale. We follow the evaluation protocol used in EffiLearner (Huang et al., 2024) to decouple *efficiency* from *correctness*

when comparing iterative optimization methods. In particular, efficiency is measured only on tasks that are already solved by DIRECT. This restriction ensures that all evaluated solutions are functionally correct, and avoids an apples-to-oranges comparison where different methods solve different subsets of tasks, making efficiency scores not directly comparable. Moreover, if an iterative method exhausts the shared 30-candidate budget without producing a valid solution, we fall back to the DIRECT solution for that task, guaranteeing an identical set of evaluated problems across methods under a fixed budget. The budget consists of five initialization candidates and 25 evolution steps; it is not 30 iterations plus 30 additional children.

Evaluation Protocol. Following our main experimental setup, each problem is evaluated on **100 test cases** with a time limit of **10 s** and a memory limit of **1024 MB**. If a candidate program triggers a runtime error, exceeds the time limit (TLE), or violates the memory limit (MLE), we mark it as a *failure* and assign it a reported metric of 0 (for ET/MP/MI).

Raw Signal for Evolution. While the paper reports the *normalized* metrics (ET/MP/MI) w.r.t. the human reference solution, the *evolution process* is guided by the **raw memory-time integral** value, denoted as A (lower is better). To reduce measurement noise, we evaluate each candidate on the same machine for **5 repeated runs**; we remove outliers by discarding the maximum and minimum runs and take the mean of the remaining runs as the final A . If any run results in failure (runtime error, TLE, or MLE), the candidate is treated as failed.

Reward and Parent Selection. Since smaller A indicates better efficiency, we convert A into a maximization-form reward for selection:

$$\mathcal{F} = \frac{1}{A},$$

We sample parents proportional to their reward, i.e.,

$$p_i = \frac{\mathcal{F}_i}{\sum_j \mathcal{F}_j}.$$

Failed candidates are assigned $\mathcal{F} = 0$, and thus will not be selected as parents. This is a standard reward-proportional sampler; we use it as an implementation choice rather than as the source of CSE’s novelty.

Search Budget and Operator Schedule. We use a shared budget of $B = 30$ candidates per task: $N_{\text{init}} = 5$ candidates are generated from diverse plans, followed by $N_{\text{evo}} = 25$ evolution steps, each generating exactly **one** child. Thus the budget is five initial candidates plus 25 children, rather than 30 iterations plus 30 children. We adopt a **deterministic alternating schedule** for the two modular evolution operators: **module-level controlled mutation** and **module-level compositional crossover** are executed in an interleaving order across iterations. Concretely, we apply mutation on odd iterations and crossover on even iterations, ensuring a stable and reproducible operator mixture throughout evolution.

Initialization and Global Memory Retrieval Hyperparameters. We use $N_{\text{init}} = 5$ diverse initialization plans per problem. For global memory retrieval, we generate $N_q = 3$ queries and retrieve top- $K_m = 3$ memories per query, resulting in at most $N_q \cdot K_m$ retrieved entries.

Adaptive Module Decomposition. To enable Modular Evolution, we use a constrained but adaptive decomposition. The LLM chooses 3–5 functional slots from the current program and execution feedback; slot names and semantics are not a fixed enum. The output schema fixes the fields `role`, `interface_constraints`, `bottleneck_diagnosis`, `observed_strengths`, and `code_span`, while the slot labels remain task-dependent. Examples such as [A] Input, [B] Core, and [C] Optimization illustrate possible descriptions rather than mandatory modules. The decomposition prompt also asks the model to attach feedback-grounded diagnoses to modules, including suspected bottlenecks, reusable strengths, and interface constraints. All subsequent mutation and crossover operations are applied at the module level, which helps maintain interface consistency and reduces unintended edits outside the targeted components.

Local Memory Compression. We maintain a local memory buffer for intra-task experiences. To prevent memory overflow in the prompt context, we perform semantic compression whenever the local memory exceeds **1000 tokens**.

Global Memory Extraction. After completing the evolution of a task τ , we convert the full task trajectory into a small set of transferable memory entries. For every evolutionary step, we record

the parent program, the generated child program, the evaluation outcome, and the reward change Δ . We then rank the trajectory by Δ and select two contrastive evidence sets: the top- K improving steps and the top- K degrading steps, with $K = 5$. These two sets, together with the task description and the final best solution y_τ^* , are passed to the global-memory extraction prompt. The extractor is instructed to compare improvements against regressions, ignore cosmetic edits, focus on strategy-level lessons such as algorithm choice, data structures, I/O patterns, caching, or memory reduction, and abstract away task-specific variable names into reusable problem-family guidance. Its output is a concise list of at most 3–5 structured experiences; each entry is marked as either a success pattern or a failure lesson and contains a short title, a one-sentence description, and a brief explanation of when and why the lesson applies. We insert these extracted entries into the global memory pool $\mathcal{M}^{\text{glb}}$ only after task τ has finished, so later tasks can retrieve from earlier-task experience while the current task does not leak future information.

Global Memory Query Generation and Retrieval. At each evolution iteration, before applying module-level mutation or crossover, CSE generates retrieval queries from the current optimization context using a dedicated query-generation prompt. The prompt receives the current problem description, constraints, language, optimization target, and accumulated local memory, then asks the model to infer likely bottlenecks and produce short natural-language retrieval queries. In our implementation, we request $N_q = 3$ queries, typically describing concrete scenarios such as fast I/O needs, pair-counting bottlenecks, or memory-heavy dynamic programming. We encode both the generated queries and all global-memory entries using **Qwen3-8B-Embedding**. For each query, we perform nearest-neighbor search over $\mathcal{M}^{\text{glb}}$ with **cosine similarity** and return the top- $K_m = 3$ entries, yielding at most nine retrieved experiences per iteration.

Use of Retrieved Memory in Context. The retrieved entries are rendered as the Global Memory field in the subsequent evolution prompt, alongside the problem statement, additional requirements, current candidate program, and the compressed Local Memory field. This means global memory is not used as a post-hoc filter; it is injected before generation and conditions the model’s decom-

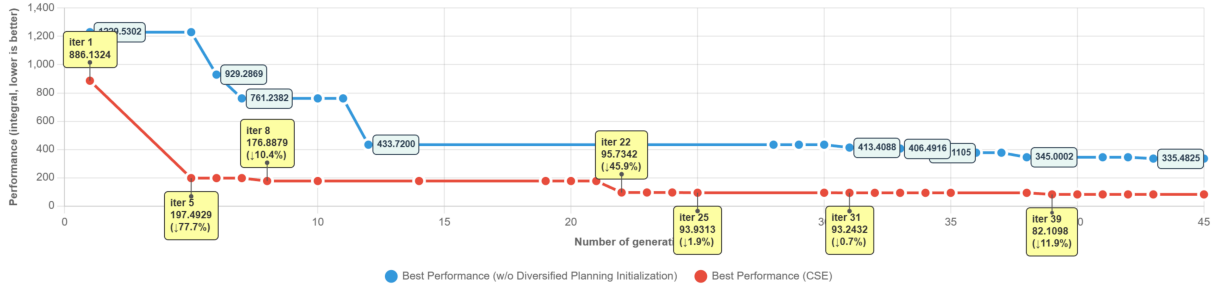
position, bottleneck diagnosis, and module-level edit decisions. Success entries serve as reusable optimization strategies to try when their scenario matches the current task, while failure entries act as negative constraints that discourage repeating previously observed regressions. The controlled-mutation and compositional-crossover prompt templates expose this context layout explicitly through the Local Memory and Global Memory fields.

Global Memory Construction Order. To ensure determinism and reproducibility, we fix a single task-processing order and apply the same order to all methods when constructing and using the global memory. After each processed task, we distill the evolved trajectory into compact experience items and add them into the global memory for retrieval in subsequent tasks.

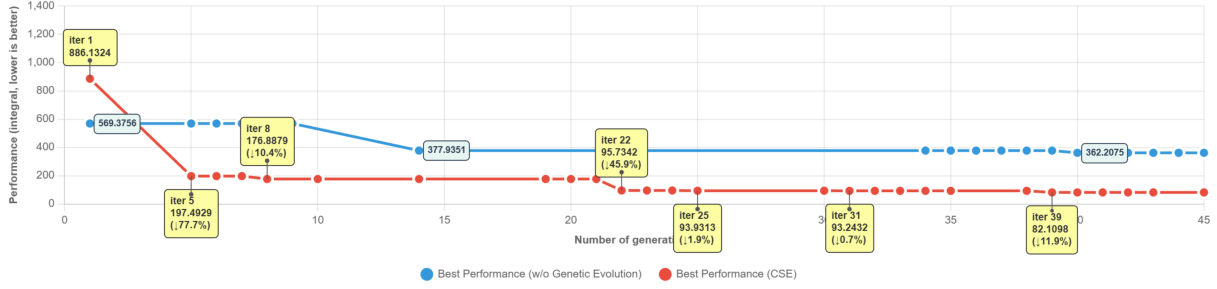
D Ablation Case Studies

Case Study: Diversified Planning Initialization.

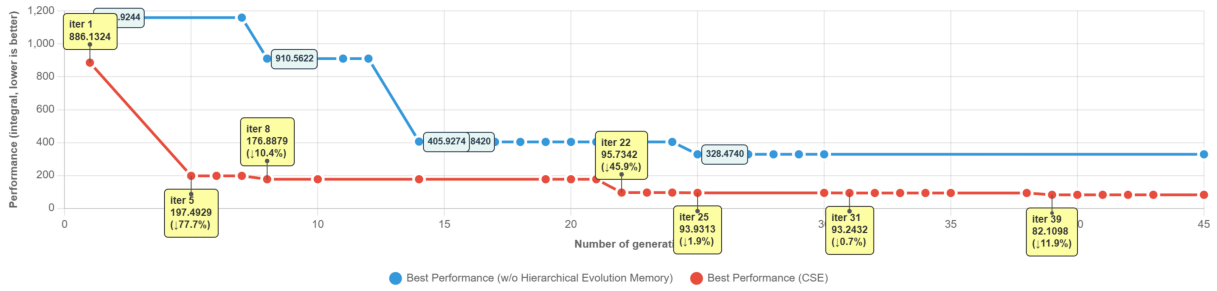
Figure 6a compares full CSE with an ablated variant that removes diversified planning initialization. We observe a clear initialization bias: without diversified planning, the search starts from a substantially worse region (best-so-far ≈ 1230.5) and remains almost unchanged for several generations, indicating that the initial candidates are clustered around a low-quality mode. Although the ablated run eventually makes progress (dropping to ≈ 929.3 and ≈ 761.2 around mid-early generations), it still struggles to escape the basin and only reaches a plateau at ≈ 433.7 after ~ 12 generations, with little improvement thereafter. In contrast, diversified planning immediately provides a much stronger and more promising starting point (iter 1: 886.1), enabling an early “algorithmic jump” within a few iterations (iter 5: 197.5, a 77.7% reduction), followed by continued refinement (iter 8: 176.9). More importantly, starting from diverse, semantically distinct strategy sketches allows the evolutionary loop to explore multiple high-potential basins in parallel, which later supports a second breakthrough (iter 22: 95.7) and a final improvement (iter 25: 93.9). Overall, removing diversified planning causes prolonged stagnation and a much higher final plateau (≈ 433.7 vs. 93.9), demonstrating that diversified planning is crucial for overcoming poor initial modes and accelerating the discovery of high-efficiency solution structures.



(a) **w/o Diversified Planning Initialization.** Best-so-far performance over generations for full CSE (red) versus removing diversified planning initialization (blue).



(b) **w/o Modular Evolution.** Best-so-far performance over generations for full CSE (red) versus removing modular evolution (blue).



(c) **w/o Hierarchical Evolution Memory.** Best-so-far performance over generations for full CSE (red) versus removing hierarchical experience memory (blue).

Figure 6: **Case studies of ablation evolution dynamics.** We compare the best-so-far **raw memory-time integral** (lower is better) across generations between full CSE (red) and three ablated variants (blue). Yellow callouts highlight representative milestone solutions along the CSE trajectory (e.g., iter 1, iter 5).

Case Study: Modular Evolution. Figure 6b illustrates the effect of removing *Modular Evolution*, i.e., the reflection-guided decomposition plus module-level mutation and composition operators. Although the ablated variant starts from a relatively strong initial point (best-so-far ≈ 569.4), it quickly becomes stagnant: the curve stays nearly flat for the first ~ 10 generations and only achieves a modest improvement to ≈ 377.9 around the mid-run, after which it plateaus again until the end of the budget. In contrast, full CSE exhibits step-wise breakthroughs characteristic of effective evolutionary search: despite a worse starting best (iter 1: 886.1), it rapidly triggers a major early jump to iter 5 (197.5; $\downarrow 77.7\%$), further refines to iter 8 (176.9), and later achieves another large improvement at iter 22 (95.7; $\downarrow 45.9\%$) followed by a final

gain at iter 25 (93.9). This gap suggests that the core benefit of Modular Evolution is not merely producing more variants, but enabling *structure-preserving, feedback-aligned* updates: module-level mutation can surgically repair the bottleneck component without disrupting already-correct parts (Eq. 2), while module-level composition can recombine complementary strengths across candidates rather than performing superficial text-level mixing (Eq. 3). Without these operators, the search lacks an effective mechanism to translate evaluation feedback into high-reward structural edits, leading to long plateaus and a much higher final performance floor (≈ 377.9 vs. 93.9).

Case Study: Hierarchical Evolution Memory. Figure 6c examines the impact of removing *Hi-*

erarchical Evolution Memory. Without memory, the search trajectory exhibits pronounced plateaus and delayed progress: the best-so-far performance remains extremely high for multiple generations (around ~ 1200), then only drops to ~ 910.6 , and later makes a larger decrease to roughly ~ 405.9 before entering a long stagnation period near ~ 400 . Even late in the run, the memory-ablated variant only achieves a modest improvement to ~ 328.5 and then plateaus again, indicating that exploration repeatedly revisits similar low-yield regions rather than systematically leveraging past lessons. In contrast, full CSE quickly consolidates early improvements (iter 5: 197.5; iter 8: 176.9) and, crucially, avoids long mid-run stagnation by triggering a late-stage breakthrough (iter 22: 95.7; $\downarrow 45.9\%$) followed by a final refinement (iter 25: 93.9). This divergence highlights the role of memory as a feedback-to-action bridge: local memory distills parent-child reward changes into actionable success patterns and failure constraints (Eq. 4), reducing redundant trial-and-error within the task, while global memory retrieval supplies reusable optimization heuristics from prior tasks that steer generation toward unexplored but promising directions (Eq. 5). Removing this mechanism substantially slows the transition from incremental tweaks to high-impact structural changes, yielding a much higher final performance floor (~ 328.5 vs. 93.9).

E Repeated-Run Stability

To assess whether the observed gains depend heavily on stochastic variation, we repeat a single CSE setting multiple times using DeepSeek-V3-0324 on 100 Python tasks. Table 5 reports summary statistics across repeated runs. The three efficiency metrics show low variation across runs, with standard deviations of 0.0201 for ET, 0.0003 for MP, and 0.0188 for MI. These results suggest that, under this setting, CSE’s outcomes are stable rather than being dominated by run-to-run randomness. The corresponding five-run comparison with SE-Agent and AlphaEvolve is reported in Table 9.

F Global Memory Ablation

To further isolate the contribution of global memory, we conduct a focused ablation with DeepSeek-V3-0324 on a 50-task subset. Both settings share the same core CSE components, and the only difference is whether retrieved global memory is injected. Under the difficulty curriculum, removing

Metric	Mean $\pm$ Std.	Min	Max	Range
ET	0.8082 ± 0.0201	0.7889	0.8379	0.0490
MP	0.9938 ± 0.0003	0.9933	0.9940	0.0007
MI	0.8617 ± 0.0188	0.8478	0.8878	0.0399

Table 5: **Repeated-run stability of CSE.** We repeat the same DeepSeek-V3-0324 CSE setting on 100 Python tasks and report mean, standard deviation, extrema, and range across runs.

Setting	MI
w/o Global Memory	1.3369
w/ Global Memory (difficulty order)	1.3454

Table 6: **Global memory ablation.** We evaluate DeepSeek-V3-0324 on 50 tasks. All settings use Diversified Planning Initialization and Modular Evolution.

global memory reduces the MI score from 1.3454 to 1.3369, a difference of $+0.0085$. The expanded order comparison in Table 13 shows why this result should be interpreted conditionally: random ordering yields an observed difference of -0.0090 .

At the same time, this global-memory mechanism is intentionally simple in our current implementation. First, we do not explicitly model the semantic relatedness among tasks; instead, we process tasks by difficulty so that useful transfer can emerge when later tasks benefit from experience distilled from earlier ones. Second, although the memory distillation step considers both successful and failed evolutionary experiences, the extracted memories are concise and only lightly constrained. They are not organized with a richer, skill-like schema that specifies reusable preconditions, failure modes, and application procedures. These design choices likely limit the observed effect size. Nevertheless, under the difficulty curriculum, this relatively coarse construction can provide a useful cross-task experience source; the random-order result indicates that the gain is modest and order-sensitive.

A natural concern is negative transfer: memories from one algorithmic family (e.g., string matching) could bias generation on an unrelated family (e.g., graph problems). In practice, because global memories serve as soft natural-language guidance and candidates are still selected by test execution, an irrelevant memory acts as prompt noise rather than a harmful constraint. The curriculum result is compatible with useful transfer, but the random-order decrease prevents a claim of consistent positive gain or order-independent absence of negative

Method	AIME 2024
Direct	13/30 (43.33%)
Self-Reflection	21/30 (70.00%)
SE-Agent	22/30 (73.33%)
AlphaEvolve	22/30 (73.33%)
CSE (Ours)	23/30 (76.67%)

Table 7: **AIME 2024 scenario extension.** We report pass counts and pass rates using DeepSeek-V3-0324 under a 30-candidate budget.

Method	Candidate pass	Fallback to Direct
Direct	100.00%	0.00%
Self-Reflection	72.10%	3.60%
SE-Agent	75.10%	4.10%
AlphaEvolve	81.80%	4.70%
CSE	79.90%	3.00%

Table 8: Candidate pass and fallback rates under the shared budget.

transfer.

G AIME 2024 Scenario Extension

To further test whether CSE can generalize beyond code-generation and code-efficiency settings, we evaluate DeepSeek-V3-0324 on AIME 2024. This benchmark provides a correctness-oriented mathematical reasoning scenario with verifiable final answers. Following the same 30-candidate budget used in our correctness-oriented code evaluation, we compare CSE against direct generation, self-reflection, SE-Agent, and AlphaEvolve.

As shown in Table 7, CSE achieves the highest pass rate on AIME 2024, improving over both SE-Agent and AlphaEvolve by one additional solved problem. The improvement is modest in absolute count, but it suggests that the controlled evolution loop can also provide useful search structure in mathematical reasoning tasks where feedback is based on final-answer correctness rather than code execution signals.

H Candidate Validity and Fallback

This appendix reports candidate validity rather than hiding execution failures in aggregate efficiency scores. The denominator is the fixed Direct-solved task set, and “Fallback” means that the method did not return a valid candidate within the shared 30-candidate budget. Table 8 summarizes the pass and fallback rates; values are from the local rebuttal records.

Method	ET	MP	MI
SE-Agent	0.8006 $\pm$ 0.0207	0.9704 $\pm$ 0.0110	0.8349 $\pm$ 0.0217
AlphaEvolve	0.8089 $\pm$ 0.0198	0.9794 $\pm$ 0.0150	0.8428 $\pm$ 0.0224
CSE	0.8082 $\pm$ 0.0201	0.9938 $\pm$ 0.0003	0.8617 $\pm$ 0.0188

Table 9: Five-run stability on 100 Python tasks.

Backbone	Comparison	Δ MI [95% CI]	Holm p
DeepSeek-V3	CSE-SE-Agent	+2.10 [0.95, 3.49]	< .001
DeepSeek-V3	CSE-AlphaEvolve	+2.55 [1.27, 4.38]	< .001
GPT-5	CSE-SE-Agent	+3.29 [1.80, 4.95]	< .001
GPT-5	CSE-AlphaEvolve	+2.79 [1.15, 4.73]	< .001

Table 10: Paired task-level MI comparisons with confidence intervals and Holm correction.

I Five-Run Stability

We repeat each method five times on the same 100 Python tasks with DeepSeek-V3-0324. Table 9 reports the mean $\pm$ standard deviation across runs.

J Paired Significance Tests

Table 10 summarizes task-level paired MI differences. Confidence intervals are paired bootstrap intervals; p values are Holm-adjusted Wilcoxon signed-rank tests. The comparisons use the same task instances for both methods.

K Matched-Token and Time-to-Quality Controls

Table 11 reports matched-token and time-to-quality comparisons, which answer different resource questions. In the matched-token setting (GPT-5, Python), the cutoff is the first point at which each method reaches the reported token budget. In the time-to-quality setting, the number of iterations is held at the reported crossing point and tokens are recorded per task. For reference, the rebuttal’s +12.10 MI comparison denotes a gain over DIRECT on Claude-4.5-Sonnet C++ (74.88 versus 62.78), not a difference from another iterative baseline.

L Held-Out Validation

Table 12 compares visible and held-out correctness for the same 100 Python tasks. We generate 50 new tests that are not exposed during evolution or best-candidate selection. “Visible” is the original test set; “held-out” is the newly generated set. MI is computed on the pass-conditioned held-out subset, so it is not a population-wide score when a candidate fails held-out correctness.

Method	Cutoff / iters.	Tokens/task	MI
<i>Matched-token (GPT-5 + Python)</i>			
AlphaEvolve	30	114,500	65.53
SE-Agent	26	110,419	64.96
CSE	25	113,651	67.92
<i>Time-to-quality</i>			
AlphaEvolve	30	114,500	65.53
SE-Agent	30	129,000	65.36
CSE	20	88,577	66.88

Table 11: Matched-token and time-to-quality comparisons. MI is reported on the same evaluation split within each comparison.

Method	Vis. pass	Held-out pass	Drop	Vis. MI	Held-out MI
Direct	100.00%	99.70%	0.30	0.5622	0.5548
Self-Reflection	100.00%	99.40%	0.60	0.6118	0.6123
SE-Agent	100.00%	100.00%	0.00	0.6069	0.6135
AlphaEvolve	99.70%	99.10%	0.60	0.6024	0.5870
CSE	100.00%	99.40%	0.60	0.6280	0.6256

Table 12: Visible versus held-out correctness and pass-conditioned MI.

M Global-Memory Ordering Boundary

Table 13 reports the order comparison on a 50-task DeepSeek-V3-0324 subset. The global-memory effect depends on task order: the difficulty curriculum yields a small positive difference, whereas random order yields a negative difference. These measurements do not support an order-independent no-negative-transfer claim.

Order	w/o memory	w/ memory	Δ MI
Difficulty curriculum	1.3369	1.3454	+0.0085
Random order	–	1.3279	−0.0090

Table 13: Order-sensitive global-memory ablation. The random-order row reports the observed w/ memory value and its difference from the corresponding no-memory condition.

All values are transcribed from the local rebuttal and experiment-design records used for this revision. They expose denominators, units, and protocol boundaries rather than turning observational ablations into causal claims.

N CSE Evolution Case Study

To demonstrate CSE in practice, Figure 7 shows the complete 30-iteration evolution on EffiBench-X ringo’s-favorite-numbers-3. The trajectory illustrates how our mechanisms work together: initialization explores diverse solution hypotheses, including a precomputation-centric strategy (Iter 1) versus a reformulated k -space search (Iter 5), showing CSE can switch high-level formulations rather

than merely refining a single approach. Once the stronger structure emerges, module-level mutations fix bottlenecks (e.g., inefficient checking or missing guards) while preserving the effective skeleton, yielding steady improvements. Module-level crossover then merges complementary strengths from different parents, such as combining robust core search with cleaner predicates. Throughout, hierarchical evolution memory guides the search away from explored variants toward novel directions, reducing redundancy. The final solution differs markedly from the human reference in both design and implementation, achieving better efficiency.

O Positioning, Scope, and Feedback Assumptions

This appendix makes the contribution boundary explicit. CSE does not claim that diversified initialization, reflection, mutation, crossover, or memory is individually new. The contribution is a control pipeline that connects these established primitives at a particular granularity: route-level diversity is converted into functional edit boundaries, execution feedback is attached to those boundaries, and local/global search state is reused to select subsequent actions.

Actionable-feedback assumption. CSE requires (i) a trustworthy correctness oracle that can reject invalid candidates and (ii) a measurable objective that can rank or compare valid candidates. The signal need not be a dense scalar and does not require a human reference solution. In our evaluations, EffiBench-X supplies correctness checks together with runtime/memory measurements, LiveCodeBench supplies hidden-test correctness, and AIME supplies final-answer verification. These settings provide an executable or otherwise checkable feedback channel for candidate selection.

Applicability boundary. When correctness is not reliably testable, when multiple outputs are valid but the evaluator cannot recognize them, or when available tests are too weak to support the claimed objective, the present execution-based loop is not sufficient. Such tasks may require a learned reward model, stronger test generation, or human feedback. We treat these settings as a scope boundary and future work rather than extrapolating the current results beyond verifiable-feedback tasks.

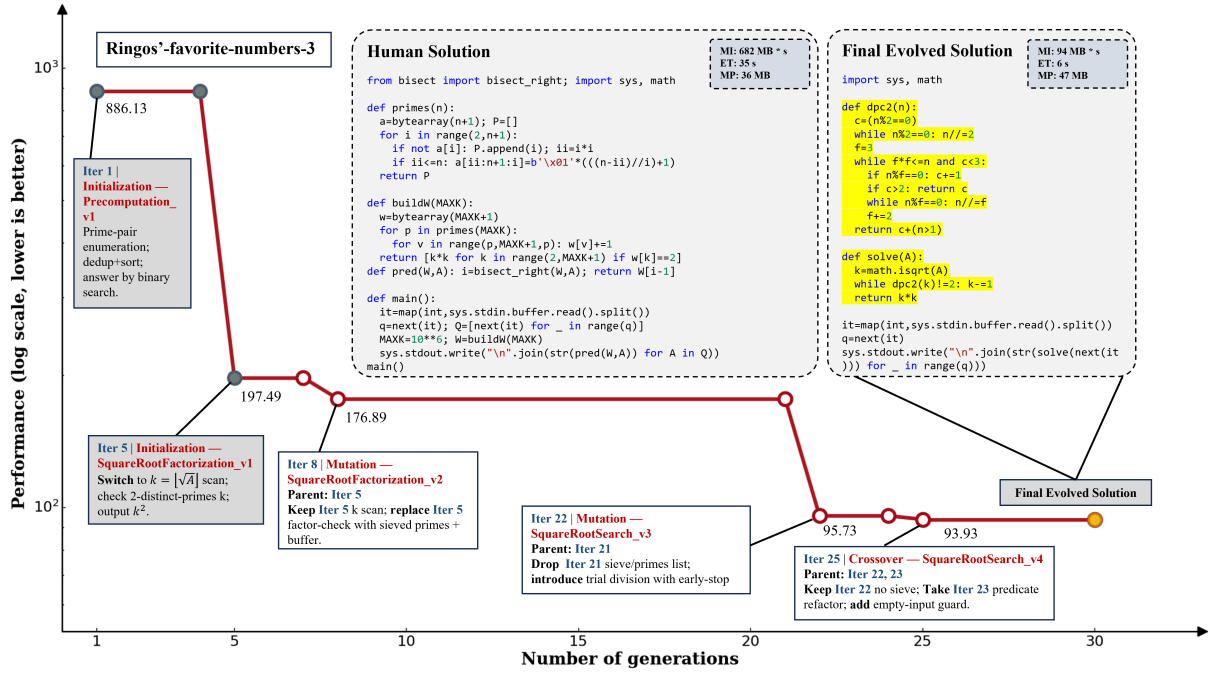


Figure 7: **A case study of CSE evolution dynamics.** To quantify progress, we plot the best-so-far **raw memory-time integral** (lower is better) against the number of generations. The case highlights the concrete logic of key initialization, module-level mutation, and module-level crossover steps, and contrasts the final evolved solution against the human solution.

Work	Route diversity	Exec. feedback	Reflection	Population variation	Module-level edit	Persistent memory	Integrated control
FunSearch	✓	✓	—	✓	×	×	×
AEL / EoH	✓	✓	—	✓	×	×	×
AlphaEvolve / OpenEvolve	✓	✓	—	✓	×	×	×
LLaMEA / LLaMEA-BO	—	✓	—	✓	×	×	×
SE-Agent	—	✓	✓	✓	×	✓	×
ReEvo	—	✓	✓	✓	×	✓	×
CSE (ours)	✓	✓	✓	✓	✓	✓	✓

Table 14: Comparison with representative LLM-driven search systems. A checkmark indicates that the capability is a reported central mechanism; “—” indicates that it is not a defining mechanism in the cited system. CSE’s novelty claim is the integrated control connection, not any individual checked primitive.

Prompt reproducibility note. The local repository contains the exact CSE prompt templates and parsing schema reproduced in Appendix P. No exact baseline-prompt source was available in the local evidence set used for this camera-ready pass; we therefore do not fabricate baseline prompts and mark their recovery as a remaining reproducibility item.

P Prompt Templates

This appendix section provides the exact system/user prompts used by our pipeline. The templates cover diversified planning to propose multiple high-performance strategies; implementation

prompts that synthesize a complete program under explicit constraints; slot-based decomposition prompts that diagnose correctness and efficiency by comparing a target solution against a reference baseline; Modular Evolution prompts for module-level controlled mutation (reflection and refine) and module-level compositional crossover (hybrid synthesis); and memory prompts that extract, compress, retrieve, and aggregate reusable success/failure experiences. Across stages, prompts emphasize correctness-first optimization, favor algorithmic improvements over micro-optimizations, and optimize the raw memory–time integral to encourage balanced runtime–memory trade-offs.

Algorithm 1 Controlled Self-Evolution (CSE)

Require: Problem specification x , Global memory $\mathcal{M}^{\text{glb}}$, $N_{\text{init}} = 5$ initial candidates, $N_{\text{evo}} = 25$ evolution steps, retrieval queries N_q , top- K size K

Ensure: Optimized solution y^*

```
1:  $\mathcal{Z} \leftarrow \text{PlanStrategies}(x)$ 
2:  $\mathcal{P}_0 \leftarrow \{y_i^{(0)} \sim \mathcal{A}_\theta(y \mid x, z_i) \mid z_i \in \mathcal{Z}\}$ 
3:  $\mathcal{M}_0^{\text{loc}} \leftarrow \emptyset$ 
4:  $\mathcal{S}_\tau \leftarrow \emptyset$ 
5: for  $t = 1$  to  $N_{\text{evo}}$  do
6:   Compute  $\{\mathcal{F}(y, x)\}_{y \in \mathcal{P}_{t-1}}$ 
7:   Sample  $y_{\text{parent}}^{(t)} \leftarrow \text{SelectParent}(\mathcal{P}_{t-1})$  with  $P_{\text{select}}(y) = \frac{\mathcal{F}(y, x)}{\sum_{y' \in \mathcal{P}_{t-1}} \mathcal{F}(y', x)}$ 
8:    $\{q_t^{(n)}\}_{n=1}^{N_q} \leftarrow \text{GenerateQueries}(\text{Context}_t)$ 
9:    $\mathcal{E}_t^{\text{ret}} \leftarrow \text{Retrieve}(\mathcal{M}^{\text{glb}}, \{q_t^{(n)}\}_{n=1}^{N_q})$ 
10:   $\text{Context}'_t \leftarrow \text{Compose}(\text{Context}_t, \mathcal{M}_{t-1}^{\text{loc}}, \mathcal{E}_t^{\text{ret}})$ 
11:   $\text{Op}_t \leftarrow \text{ChooseOp}(\text{ModularRefine}, \text{ModularCompose})$ 
12:  if  $\text{Op}_t = \text{ModularRefine}$  then
13:     $(\{c_i\}, r_t) \leftarrow \text{Decompose}(y_{\text{parent}}^{(t)} \mid \text{Context}'_t)$ 
14:     $y_{\text{child}}^{(t)} \leftarrow \text{ModularRefine}(\{c_i\}, r_t)$ 
15:  else
16:    Sample  $y_{\text{parent}}'^{(t)} \leftarrow \text{SelectParent}(\mathcal{P}_{t-1})$  with  $P_{\text{select}}(\cdot)$ 
17:     $(\{c_i\}, \{c'_i\}, r_t) \leftarrow \text{Decompose}(y_{\text{parent}}^{(t)}, y_{\text{parent}}'^{(t)} \mid \text{Context}'_t)$ 
18:     $y_{\text{child}}^{(t)} \leftarrow \text{ModularCompose}(\{c_i\}, \{c'_i\}, r_t)$ 
19:  end if
20:   $\Delta_t \leftarrow \mathcal{F}(y_{\text{child}}^{(t)}, x) - \mathcal{F}(y_{\text{parent}}^{(t)}, x)$ 
21:   $\mathcal{P}_t \leftarrow \text{UpdatePopulation}(\mathcal{P}_{t-1}, y_{\text{child}}^{(t)}, \mathcal{F}(y_{\text{child}}^{(t)}, x))$ 
22:   $m_t \leftarrow \text{Reflect}(y_{\text{child}}^{(t)}, y_{\text{parent}}^{(t)}, \Delta_t)$ 
23:   $\mathcal{M}_t^{\text{loc}} \leftarrow \text{Compress}(\mathcal{M}_{t-1}^{\text{loc}} \cup \{m_t\})$ 
24:   $\mathcal{S}_\tau \leftarrow \mathcal{S}_\tau \cup \{(y_{\text{parent}}^{(t)}, y_{\text{child}}^{(t)}, \Delta_t)\}$ 
25: end for
26:  $y^* \leftarrow \arg \max_{y \in \bigcup_{t=0}^{N_{\text{evo}}} \mathcal{P}_t} \mathcal{F}(y, x)$ 
27:  $\mathcal{S}_\tau^+ \leftarrow \text{TopK}_{\max \Delta}(\mathcal{S}_\tau, K)$ 
28:  $\mathcal{S}_\tau^- \leftarrow \text{TopK}_{\min \Delta}(\mathcal{S}_\tau, K)$ 
29:  $g_\tau \leftarrow \text{Reflect}(\mathcal{S}_\tau^+, \mathcal{S}_\tau^-, y^*)$ 
30:  $\mathcal{M}^{\text{glb}} \leftarrow \mathcal{M}^{\text{glb}} \cup \{g_\tau\}$ 
31: return  $y^*$ 
```

System Prompt

You are a world-class Algorithm Engineer. Design EXACTLY K distinct, high-performance strategies for the given problem.

Guidelines:

1. ****Diversity****: Strategies MUST differ in algorithmic paradigms (DP, Greedy, Two Pointers, Bit Manipulation, etc.) or core data structures.
2. ****Performance****: Prioritize optimal Time/Space complexity. Avoid brute-force unless necessary.
3. ****Content****: Each strategy must include:
 - Core algorithmic logic
 - Key data structures
 - Expected Time/Space complexity (Big-O)

Output Format:

```
```json
{"strategies": ["<strategy_1>", "<strategy_2>", ...]}
```
```

The array must contain exactly K strings. NO extra text.

User Prompt

Problem Description:

{problem_text}

Required Strategy Count: {k}

Figure 8: Prompts for Diversified Planning Initialization.

System Prompt

You are an expert competitive programmer specializing in algorithm optimization.

Principles

1. ****Correctness First****: Ensure correctness before optimizing for {optimization_target}.
2. ****Algorithmic Focus****: Prefer algorithmic improvements over micro-optimizations. Explore directions ****different from existing approaches****.
3. ****Learn from Failures****: Use Memory to prune failed directions, but don't restrict creative exploration.
4. ****Strict Improvement****: Your solution must outperform the best prior solution in 'Additional Requirements'.

Task

Synthesize a ****complete program in {language}**** that:

- * Fixes fundamental bottlenecks in current solutions
- * Explores novel optimization directions
- * Minimizes the ****integral**** (area under memory-time curve) across all test cases -- balancing both runtime and memory

Problem

{task_description}

Response Format

1. Thinking

- * ****Baseline Analysis****: Identify strategy mode and best baseline from 'Additional Requirements'.

```

* **Failure Pruning**: Review failed directions from Memory; avoid repeating them.
* **New Direction**: Select a fundamentally different, high-potential approach. Justify why it's
  better for {optimization_target}.
* **Risks**: Note correctness concerns and trade-offs.

### 2. Final Code

Complete, self-contained {language} program. No test harness or debug prints.

## Allowed Imports

{allowed_imports_scope}

## Context

**Additional Requirements**:

### STRATEGY MODE: PLAN STRATEGY

You must strictly follow and implement the outlined approach below.

{strategy}

**Local Memory**: {local_memory}
**Global Memory**: {global_memory}

```

User Prompt

Using all context (Additional Requirements, Local/Global Memory, current program), generate ‘## Thinking’ and ‘## Final Code’ to improve PERFORMANCE. Follow the guidance from Memory.

Figure 9: Prompts for Implement Sketches.

System Prompt

You are a code diagnosis agent. Analyze ****Target Solution**** for a {language} problem, decompose into slots, and diagnose correctness/performance vs ****Reference Solution****.

****Optimization target****: {optimization_target} (integral = area under memory-time curve). Balance runtime and memory.

Task

- **Summarize****: Nickname + high-level strategy
- **Factorize****: infer 3--5 functional slots from the target program and execution feedback. Slot names and semantics are adaptive; examples include input handling, core computation, edge-case handling, or performance patches but are not a fixed enum.
- **Diagnose****: Identify bottlenecks, risks, good parts to inherit

Output JSON

```

““json
{
  "solution_name": "String",
  "approach_summary": "String",
  "slot_view": {
    "slots": [{"slot_id": "task_defined_string", "role": "String", "description": "String", "
      interface_constraints": ["..."], "tags": ["impl_method"], "code_span": [start, end]],
    "diagnoses": [{"slot_id": "String", "bottleneck_diagnosis": "String", "observed_strengths":
      ["..."], "status": "ok|bottleneck|bug_source|risky|redundant", "correctness_level": "good|
      minor_risk|major_risk|unknown", "perf_level": "strong|weak|neutral|unknown", "priority": "
      inherit|optimize|inspect|low", "evidence": ["..."]}]}

```

```

}
}
'''

### Rules

* tags: Use specific impl methods (e.g., "segment_tree", "fast_io"), not generic descriptions
* perf_level: Compare complexity vs Reference ( $O(N^2)$  vs  $O(N)$  -> weak)
* priority: inherit=keep, optimize=bottleneck, inspect=risky
* Self-contained evidence: Be specific (e.g., " $O(N^2)$  nested loop" not "slower than Reference")

Return JSON only (can wrap in '''json fence).

```

User Prompt

Diagnose the Target Solution vs Reference.

Problem

{problem_description}

Reference Solution (Baseline)

{best_solution}

Target Solution

{target_solution}

Generate JSON: decompose into slots, assign tags, set perf_level vs Reference, set priority (optimize bottlenecks, inherit good parts).

Figure 10: Prompts for Decomposition (slot-based diagnosis).

System Prompt

You are an expert competitive programmer specializing in algorithm optimization.

Principles

1. **Correctness First**: Ensure correctness before optimizing for {optimization_target}.
2. **Algorithmic Focus**: Prefer algorithmic improvements over micro-optimizations. Explore directions **different** from existing approaches.
3. **Learn from Failures**: Use Memory to prune failed directions, but don't restrict creative exploration.
4. **Strict Improvement**: Your solution must outperform the best prior solution in 'Additional Requirements'.

Task

Synthesize a **complete** program in {language} that:

- * Fixes fundamental bottlenecks in current solutions
- * Explores novel optimization directions
- * Minimizes the **integral** (area under memory-time curve) across all test cases -- balancing both runtime and memory

Problem

{task_description}

```

## Response Format

### 1. Thinking

* Baseline Analysis: Identify strategy mode and best baseline from 'Additional Requirements'.
* Failure Pruning: Review failed directions from Memory; avoid repeating them.
* New Direction: Select a fundamentally different, high-potential approach. Justify why it's
    better for {optimization_target}.
* Risks: Note correctness concerns and trade-offs.

### 2. Final Code

Complete, self-contained {language} program. No test harness or debug prints.

## Allowed Imports

{allowed_imports_scope}

## Context

Additional Requirements:

### STRATEGY MODE: REFLECTION AND REFINE

You must reflect on the previous solution and implement targeted improvements.

### SOURCE SOLUTION SUMMARY

{source_summary}

### REFINEMENT GUIDELINES

1. Diagnose: Identify main shortcomings (correctness risks, performance bottlenecks, redundant
    work, I/O overhead).
2. Targeted Fixes: Apply code-level changes to the faulty component ONLY:

    * Algorithmic upgrade for the specific bottleneck
    * Data structure replacement for the problematic module
    * Caching/precomputation for repeated work
3. Preserve Working Parts: Keep the remaining well-performing components frozen.
4. Correctness First: Ensure correctness before optimizing runtime.

Local Memory: {local_memory}
Global Memory: {global_memory}

```

User Prompt

Using all context (Additional Requirements, Local/Global Memory, current program), generate '## Thinking' and '## Final Code' to improve PERFORMANCE. Follow the guidance from Memory.

Figure 11: Prompts for Module-Level Controlled Mutation.

System Prompt

You are an expert competitive programmer specializing in algorithm optimization.

Principles

1. **Correctness First**: Ensure correctness before optimizing for {optimization_target}.
2. **Algorithmic Focus**: Prefer algorithmic improvements over micro-optimizations. Explore directions **different from existing approaches**.

```

3. **Learn from Failures**: Use Memory to prune failed directions, but don't restrict creative exploration.
4. **Strict Improvement**: Your solution must outperform the best prior solution in 'Additional Requirements'.

## Task

Synthesize a **complete program in {language}** that:

* Fixes fundamental bottlenecks in current solutions
* Explores novel optimization directions
* Minimizes the **integral** (area under memory-time curve) across all test cases -- balancing both runtime and memory

## Problem

{task_description}

## Response Format

### 1. Thinking

* **Baseline Analysis**: Identify strategy mode and best baseline from 'Additional Requirements'.
* **Failure Pruning**: Review failed directions from Memory; avoid repeating them.
* **New Direction**: Select a fundamentally different, high-potential approach. Justify why it's better for {optimization_target}.
* **Risks**: Note correctness concerns and trade-offs.

### 2. Final Code

Complete, self-contained {language} program. No test harness or debug prints.

## Allowed Imports

{allowed_imports_scope}

## Context

**Additional Requirements**:

### STRATEGY MODE: CROSSOVER

Synthesize a SUPERIOR hybrid solution by combining the best elements of two prior trajectories.

### SOLUTION 1 SUMMARY

{solution_1}

### SOLUTION 2 SUMMARY

{solution_2}

### SYNTHESIS GUIDELINES

1. **Complementary Combination**: Actively combine specific strengths from each solution.

    * If T1 has better core algorithm but slow I/O, and T2 has fast I/O but weaker algorithm -> implement T1's algorithm with T2's I/O.
    * If T1 has correct logic but slow structure, and T2 has fast structure but buggy logic -> implement T1's logic with T2's structure.
2. **Avoid Shared Weaknesses**: If both failed at a specific sub-task, introduce a novel fix.
3. **Seamless Integration**: The result must be a single, cohesive implementation--not concatenated code.

**Local Memory**: {local_memory}
**Global Memory**: {global_memory}

```

User Prompt

Using all context (Additional Requirements, Local/Global Memory, current program), generate ‘## Thinking’ and ‘## Final Code’ to improve PERFORMANCE. Follow the guidance from Memory.

Figure 12: Prompts for Module-Level Compositional Crossover.

System Prompt

You observed an evolutionary step where metrics **improved**. Extract strategy-level insights.

Goal:

1. Decide if this is truly **Success** or just **Neutral** (noise/trivial refactor).
2. If strategy-level changes exist, extract up to 3:

- * **Direction items**: Reusable optimization strategies
- * **Memory items**: Reasoning patterns explaining WHY it works

Strategy-level changes include: algorithm switch, data structure change, I/O optimization, major loop restructuring.

Do NOT create items for: variable renaming, formatting, measurement noise.

Output Format:

```
```json
{
 "thought_process": "Brief reasoning.",
 "new_direction_items": [
 {"direction": "Strategy name", "description": "When/why to use.", "status": "Success | Neutral"}
],
 "new_memory_items": [
 {"type": "Success", "title": "Technique name", "description": "One-sentence summary", "content": "2-6 sentences on when/why it works."}
]
}
```

### User Prompt

## Source Solutions

{source\_solutions}

## Current Solution

{current\_solution}

## Current Directions (Strategy Board)

{directions}

Figure 13: Prompts for Local Memory Extract (Success).

### System Prompt

You observed an evolutionary step where metrics **\*\*regressed\*\*** or correctness broke. Extract warnings.

Goal:

1. Decide if this is truly **\*\*Failure\*\*** or just **\*\*Neutral\*\*** (noise).
2. If strategy-level changes caused the regression, extract up to 3:

- \* **\*\*Direction items\*\***: Strategies to mark as Failed/risky
- \* **\*\*Memory items\*\***: Anti-patterns explaining WHY it failed

For Failure memory items:

- \* Title: Describe the SPECIFIC mistake (e.g., "BFS without boundary check causes OOB"), not just "BFS implementation"
- \* Content: Explain what went wrong and what condition triggered it

Output Format:

```
```json
{
  "thought_process": "Brief reasoning.",
  "new_direction_items": [
    {"direction": "Failed strategy", "description": "Why problematic.", "status": "Failed | Neutral"}
  ],
  "new_memory_items": [
    {"type": "Failure", "title": "Avoid ...", "description": "Why dangerous", "content": "What went wrong, how to avoid."}
  ]
}
```

User Prompt

Source Solutions

{source_solutions}

Current Solution (Failed)

{current_solution}

Current Directions (Strategy Board)

{directions}

Figure 14: Prompts for Local Memory Extract (Failure).

System Prompt

Compress the ****direction_board**** of an evolutionary agent.

Rules:

1. ****Merge similar strategies****: Combine entries describing the same idea.
2. ****Do NOT merge different failure modes****: Keep separate if root causes differ.
3. ****Aggregate counts****: When merging, SUM success_count and failure_count; update status accordingly.

4. ****Prune low-value entries****: Remove vague or noise entries. Keep ~5-10 useful directions.

Output Format:

```
```json
{
 "thought_process": "Brief explanation.",
 "direction_board": [
 {"direction": "Strategy", "description": "Explanation", "status": "Success|Failed|Neutral", "
 success_count": N, "failure_count": M}
]
}
```
```

Compress the ****experience_library**** of an evolutionary agent.

Rules:

1. ****Merge overlapping experiences****: Combine entries describing the same lesson.
2. ****Do NOT merge different root causes****: "Avoid recursion without memoization (TLE)" and "Avoid large array (OOM)" are DIFFERENT.
3. ****Content Guidelines****:

- * Success: Explain WHY it works, under what conditions
- * Failure: Describe the SPECIFIC mistake and what triggered it

4. ****Filter trivial items****: Remove noise entries. Keep ~5-8 useful experiences.

Output Format:

```
```json
{
 "thought_process": "Brief explanation.",
 "experience_library": [
 {"type": "Success|Failure", "title": "Specific title", "description": "One-sentence summary",
 "content": "2-6 sentences on when/why."}
]
}
```
```

User Prompt

Directions (Strategy Board)

{directions}

Experiences

{experience_library}

Figure 15: Prompts for Local Memory Compress.

System Prompt

Design search queries for a global memory of past optimization experiences.

Your job:

1. Read the context (problem, constraints, language, optimization target, local memory).
2. Think about potential bottlenecks in this task.
3. Output 2-3 concrete queries to search for useful experiences.

Query format: Short natural-language descriptions of scenario + problem, e.g.:

- * "In Python, how to handle fast IO when reading $2e5$ integers"
- * "For pair counting with N up to $2e5$, how to avoid $O(N^2)$ loops"
- * "How to reduce DP memory when table is $O(N^2)$ and $N=5000$ "

Output Format:

```
```json
{
 "thought_process": "Brief analysis of key bottlenecks.",
 "queries": ["<query_1>", "<query_2>", "<optional_query_3>"]
}
```
```

User Prompt

Problem Description

{problem_description}

Optimization Target

{optimization_target}

Language

{language}

Local Memory

{local_memory}

Figure 16: Prompts for Query Generation (memory retrieval).

System Prompt

Extract generalizable experiences from optimization trajectories for the SAME problem.

You will receive:

- * **Improvement Steps**: Changes that improved metrics
- * **Regression Steps**: Changes that worsened metrics
- * **Best Solution**: Current best solution

Goal: Build an experience library with Success/Failure lessons reusable on future tasks.

Guidelines:

1. **Contrastive reasoning**: Compare improvements vs regressions. What consistently works/fails?
2. **Strategy-level focus**: Algorithm choice, data structure, I/O patterns, caching. Ignore cosmetic changes.
3. **Generalize**: Abstract away specific names. Describe for problem families (e.g., "pair counting with constraints", "DP with large N ").
4. **Limit**: Output at most 3-5 experiences. Prioritize high-impact, clearly explainable changes.

Output Format:

```
```json
{
 "thought_process": "Brief contrastive analysis.",

```

```
"experiences": [
 {
 "type": "Success | Failure",
 "title": "For Success: technique name. For Failure: 'Avoid ...'",
 "description": "One-sentence summary.",
 "content": "1-5 sentences on when/why this works or fails."
 }
]
}'''
```

### User Prompt

```
Problem Description
{problem_description}
Improvement Steps
{improvement_steps}
Regression Steps
{regression_steps}
Best Solution
{best_solution}
```

Figure 17: Prompts for Global Memory Extract.