

Controlled Self-Evolution for Algorithmic Code Optimization

Anonymous ACL submission

Abstract

Self-evolution methods enhance code generation through iterative “generate-verify-refine” cycles, yet existing approaches suffer from low exploration efficiency, failing to discover solutions with superior complexity within limited budgets. This inefficiency stems from initialization bias trapping evolution in poor solution regions, uncontrolled stochastic operations lacking feedback guidance, and insufficient experience utilization across tasks. To address these bottlenecks, we propose Controlled Self-Evolution (CSE), which consists of three key components. Diversified Planning Initialization generates structurally distinct algorithmic strategies for broad solution space coverage. Genetic Evolution replaces stochastic operations with feedback-guided mechanisms, enabling targeted mutation and compositional crossover. Hierarchical Evolution Memory captures both successful and failed experiences at inter-task and intra-task levels. Experiments on EffiBench-X demonstrate that CSE consistently outperforms all baselines across various LLM backbones. Furthermore, CSE achieves higher efficiency from early generations and maintains continuous improvement throughout evolution.

1 Introduction

Code generation has emerged as a critical application of Large Language Models (LLMs) (Comanici et al., 2025; Achiam et al., 2023; Cai et al., 2024; Dubey et al., 2024; Team et al., 2025; Guo et al., 2025), with models demonstrating impressive capabilities in producing functionally correct solutions for programming tasks (Chen, 2021; Jiang et al., 2024; Dong et al., 2025; Wang et al., 2025a; Hui et al., 2024; Wang et al., 2024, 2025b). Early approaches relied on single-turn generation, where models directly produce complete solutions from problem specifications (Luo et al., 2023; Wei et al., 2024). While achieving reasonable success on simple tasks, this paradigm struggles with complex

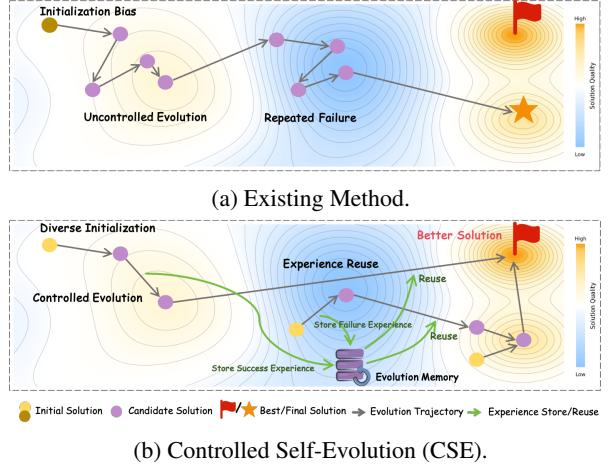


Figure 1: **Controlled Self-Evolution (CSE) improves exploration efficiency.** (a) Existing self-evolution wastes budget on low-quality regions due to initialization bias, uncontrolled evolution and repeated failure. (b) CSE guides exploration toward higher-quality solutions through diversified initialization, controlled evolution, and experience reuse.

algorithmic problems and lacks mechanisms to interact with execution environments or leverage verification feedback. To address these limitations, self-evolution methods (Gao et al., 2025; Fang et al., 2025) have emerged as a promising paradigm that enables iterative “generate-verify-refine” cycles: models start from initial solutions, execute code against test cases, analyze feedback signals (e.g., failed tests, performance bottlenecks), and generate improved variants. Methods such as AlphaEvolve (Novikov et al., 2025) and SE-Agent (Lin et al., 2025) have demonstrated that treating code generation as a feedback-driven search process can substantially enhance code quality.

Ideally, with unlimited computational resources, we could allow these self-evolving agents to explore extensively until reaching optimal solutions that are not only functionally correct but also exhibit superior time and space complexity. How-

ever, practical deployment scenarios impose strict resource constraints. In real-world applications, extensive multi-turn inference incurs prohibitive computational costs and latency. This creates a fundamental tension: we need methods that can discover high-quality solutions, measured not just by correctness but by algorithmic efficiency, within a limited exploration budget.

Unfortunately, existing self-evolution methods suffer from critically low exploration efficiency. This inefficiency prevents them from discovering code with superior time and space complexity within limited exploration budgets, as the search process fails to efficiently navigate toward algorithmically optimal solutions. This inefficiency stems from three fundamental limitations (as shown in Figure 1). First, *initialization bias*: methods (Du et al., 2025; Huang et al., 2024) typically begin evolution from a single or few initial solutions generated by the base model, which may lie in poor regions of the solution space, necessitating many iterations to escape local optima. Second, *uncontrolled stochastic evolution*: operations like random mutation and crossover are applied without explicit guidance from feedback signals, resulting in undirected exploration where many generated variants fail to systematically navigate toward better solutions. Third, *insufficient utilization of evolution experience*: existing approaches (Lin et al., 2025; Du et al., 2025) do not effectively accumulate successful patterns within a task or abstract transferable experiences across tasks, causing repeated failures and preventing the reuse of proven optimization strategies.

This motivates our central research question: **Can we design a self-evolution framework that achieves high code quality while dramatically improving exploration efficiency?** We propose **Controlled Self-Evolution (CSE)**, a novel framework that addresses all three efficiency bottlenecks through three key innovations. First, Diversified Planning Initialization generates multiple structurally distinct algorithmic strategies before evolution begins, ensuring broad coverage of the solution space and reducing the likelihood of getting trapped in poor local regions. Second, Genetic Evolution replaces stochastic operations with fine-grained feedback-guided mechanisms: functional decomposition enables targeted mutation that refines faulty components while preserving high-performing parts, and compositional crossover structurally integrates complementary

strengths from diverse solutions. Third, Hierarchical Evolution Memory captures and reuses evolutionary insights at two levels: local memory accumulates task-specific lessons to avoid repeating failures within the current problem, while global memory distills cross-task optimization patterns into reusable templates that accelerate future evolution. We conduct comprehensive experiments on EffiBench-X(Qing et al., 2025), showing that CSE consistently outperforms all baselines across diverse LLM backbones, achieves stronger efficiency from early generations, and continues improving throughout evolution. These results demonstrate robust, backbone-agnostic gains with both fast-starting and sustained progress.

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.; Lavon et al., 2025; Yao et al., 2025). Recent advances span instruction tuning (WizardCoder (Luo et al., 2023), Magicoder (Wei et al., 2024)), retrieval augmentation (Lu et al., 2022), while specialized models such as GPT-4o (Hurst et al., 2024) and DeepSeekCoder (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., 2025) 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.

Self-Evolution. To address single-turn generation limitations, self-evolution methods enable iterative refinement through "generate-verify-refine" cycles. Self-reflection (Madaan et al., 2023; Shinn et al., 2023; Du et al., 2025; Sun et al., 2023) approaches enable models to learn from execution feedback, though they focus on debugging rather than algorithmic optimization. Such as AfterBurner (Du et al., 2025) which perform local refinement but suffer from initialization bias and local optima. Population-based approaches (Romera-Paredes et al., 2023; Liu et al., 2024b; Lange et al., 2025) such as AlphaEvolve (Novikov

et al., 2025) employ evolutionary resampling with stochastic mutations, while SE-Agent (Lin et al., 2025) introduces trajectory-level evolution via step-wise recombination. However, these methods face low search efficiency due to unguided exploration through random mutations, and lack of experience reuse. They thus require extensive iterations and often fail to discover optimal solutions within limited budgets. Our CSE framework addresses this through diversified initialization, feedback-guided operations, and hierarchical memory for efficient, controlled evolution.

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 exploration rather than uncontrolled stochastic evolution.

4 Method

As illustrated in Figure 2, our method consists of three key components: **Diversified Planning Initialization**, **Genetic Evolution**, and **Hierarchical Evolution Memory**. Diversified Planning Initialization generates high-quality and diverse initial solutions through varied planning strategies, providing a strong foundation for subsequent evolution. Unlike prior black-box evolutionary approaches, Genetic Evolution enables fine-grained, controllable mutation to efficiently navigate toward optimal solutions. Hierarchical Evolution Memory

hierarchically summarizes both intra-task and inter-task experiences, providing search guidance for the evolutionary process. In the following subsections, we describe each component in detail.

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}_θ 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. For example, given a code generation task, the agent might propose fundamentally different approaches such as a greedy algorithm, dynamic programming, or bit manipulation optimization, rather than superficial variations of the same logic.

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

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

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 $\mathcal{P}_0 = \{y_1^{(0)}, \dots, y_{N_{\text{init}}}^{(0)}\}$ is no longer composed of random perturbations around a single mode, but instead achieves diverse coverage of the solution space. This ensures that the subsequent evolutionary process can explore multiple promising regions in parallel, significantly reducing the risk of premature convergence to a suboptimal local optimum.

4.2 Genetic Evolution

The goal of self-evolutionary update is to guide the model in searching for better trajectories based on existing ones. A key insight is that incorporating explicit control into the search process, rather than relying on uncontrolled stochastic evolution, can significantly improve search efficiency. To this end, we design a Genetic Evolution mechanism with the following components:

Parent Selection. We depart from prior evolutionary approaches (Lin et al., 2025) that exclusively select high-reward solutions, as low-reward solutions may also contain valuable parts. We design a probability-based parent selection strategy

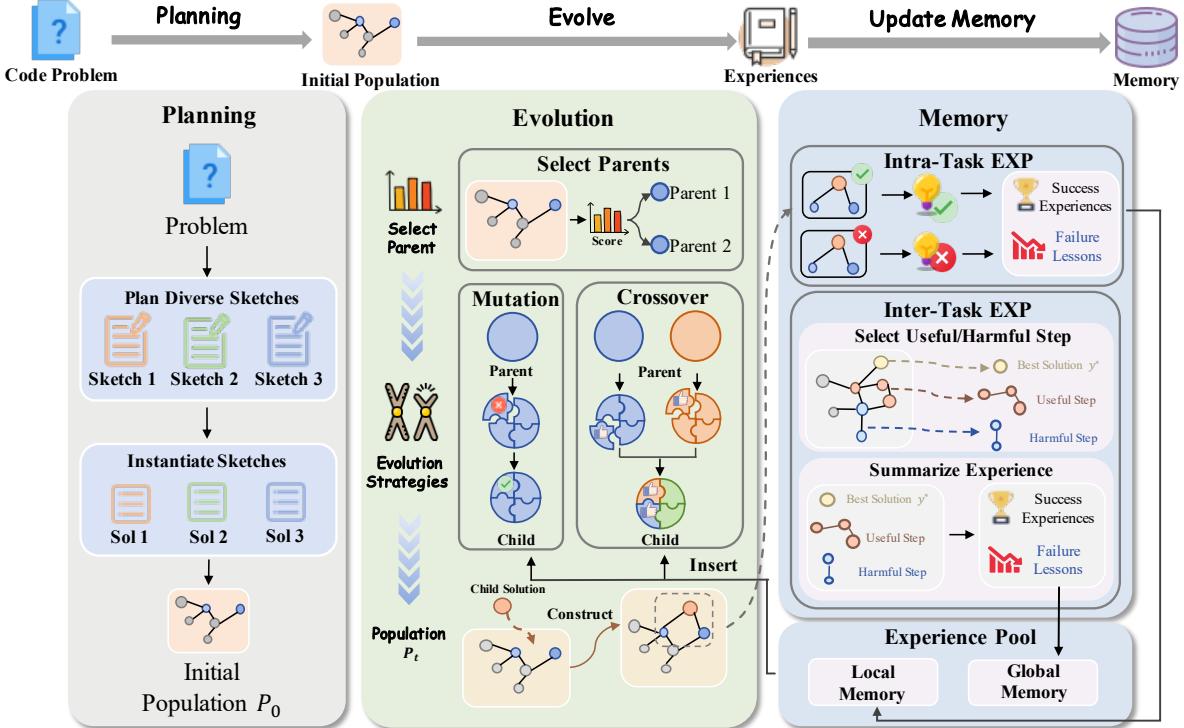


Figure 2: **Overview of the CSE.** Our method consists of three key components: Diversified Planning Initialization, Genetic Evolution, and Hierarchical Evolution Memory.

where the selection probability of each candidate is proportional to its normalized reward:

$$P_{\text{select}}(y_i^{(t)}) = \frac{\mathcal{F}(y_i^{(t)}, x)}{\sum_{j=1}^{|P_t|} \mathcal{F}(y_j^{(t)}, x)}, \quad (2)$$

where $P_{\text{select}}(y_i^{(t)})$ denotes the selection probability of candidate solution $y_i^{(t)}$, and P_t represents the population at evolution step t . This mechanism implements a soft selection distribution that prioritizes high-reward individuals while retaining lower-reward individuals that may contain useful logic fragments or partial solutions for recombination.

Evolution Strategies. To enable fine-grained controlled evolution, we first prompt the agent to decompose the code y into a set of disjoint functional components $\{c_1, c_2, \dots, c_m\}$ (e.g., I/O parsing module, core algorithm logic, boundary condition handling). This functional decomposition provides the necessary structural foundation for subsequent fine-grained interventions. We then equip the agent with two controlled evolution strategies:

Controlled Mutation. The agent employs self-reflection to identify the specific faulty component c_{faulty} responsible for low reward. After localizing the problematic module, we perform targeted regeneration on c_{faulty} while keeping the remaining

$m - 1$ well-performing components frozen:

$$y_{\text{child}} = \{c_1, \dots, \text{Refine}(c_{\text{faulty}}), \dots, c_m\}. \quad (3)$$

This surgical repair strategy not only avoids disruptive interference with the global context but also significantly improves mutation efficiency.

Compositional Crossover. To facilitate the flow of advantageous traits across the population, we introduce compositional crossover, which performs logic-level recombination of strengths rather than naive textual concatenation. Suppose parent solutions y_A and y_B exhibit complementary advantages in different components (e.g., y_A has superior time complexity while y_B demonstrates better robustness). The crossover operator synthesizes these strengths structurally:

$$y_{\text{child}} = \text{Crossover}(\{c_{\text{time}}^{(A)}\}, \{c_{\text{robust}}^{(B)}\}), \quad (4)$$

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 . This enables the agent to mimic the human practice of combining complementary strengths from different solutions.

307 4.3 Hierarchical Evolution Memory

308 To effectively leverage both intra-task and inter-
 309 task experiences, we propose Hierarchical Evolu-
 310 tion Memory.

311 **Local Memory.** $\mathcal{M}_{\text{local}}$ aims to capture imme-
 312 diate experiences from intra-task search trajectories.
 313 At each evolutionary step t , we compare the re-
 314 ward change $\Delta_t = \mathcal{F}(y_{\text{child}}^{(t)}, x) - \mathcal{F}(y_{\text{parent}}^{(t)}, x)$
 315 between the parent $y_{\text{parent}}^{(t)}$ and its child $y_{\text{child}}^{(t)}$. The
 316 agent extracts two types of critical experiences:
 317 success insights ($\Delta_t > 0$) that analyze what led to
 318 improvements and are marked as positive patterns
 319 to retain, and failure lessons ($\Delta_t \leq 0$) that ana-
 320 lyze causes of reward degradation and are marked
 321 as negative constraints to avoid. This reflection
 322 process is formalized as:

$$323 m_t \leftarrow \text{Reflect}\left(y_{\text{child}}^{(t)}, y_{\text{parent}}^{(t)}, \Delta_t\right), \quad (5a)$$

$$324 \mathcal{M}_t^{\text{loc}} \leftarrow \text{Compress}(\mathcal{M}_{t-1}^{\text{loc}} \cup \{m_t\}), \quad (5b)$$

325 where m_t denotes the distilled experience at step
 326 t . At evolution step $t+1$, the accumulated $\mathcal{M}_{\text{local}}$
 327 is dynamically injected into the prompt context,
 328 forming bidirectional guidance: success insights
 329 explicitly instruct the model to reuse validated op-
 330 timization strategies, while failure lessons prevent
 331 the model from repeating known mistakes. To pre-
 332 vent memory overflow as iterations accumulate,
 333 whenever $\mathcal{M}_t^{\text{loc}}$ exceeds a predefined length thresh-
 334 old, we perform semantic compression to maintain
 335 high information density in the local memory.

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

$$344 g_\tau \leftarrow \text{Reflect}(\mathcal{S}_\tau^+, \mathcal{S}_\tau^-, y_\tau^*), \quad (6a)$$

$$345 \mathcal{M}^{\text{glb}} \leftarrow \mathcal{M}^{\text{glb}} \cup \{g_\tau\}. \quad (6b)$$

346 At evolutionary step t , the agent generates N_q tar-
 347 geted retrieval queries based on the current evolu-
 348 tionary context (including the current code state,
 349 encountered errors, or performance bottlenecks):

$$350 \{q_t^{(n)}\}_{n=1}^{N_q} \leftarrow \text{GenerateQueries}(\text{Context}_t), \quad (7a)$$

$$351 \mathcal{E}_t^{\text{ret}} \leftarrow \text{Retrieve}(\mathcal{M}^{\text{glb}}, \{q_t^{(n)}\}_{n=1}^{N_q}), \quad (7b)$$

352 where $\{q_t^{(n)}\}_{n=1}^{N_q}$ represents the generated queries
 353 and $\mathcal{E}_t^{\text{ret}}$ denotes the retrieved relevant experiences.
 354 This mechanism ensures that the agent can pre-
 355 cisely invoke past experiences from similar tasks
 356 when encountering specific challenges.

357 As summarized in Algorithm 1, CSE first gener-
 358 ates diverse strategy sketches \mathcal{Z} and instantiates
 359 them into an initial population \mathcal{P}_0 with \mathcal{A}_θ . It
 360 then iterates for T steps: at iteration t , it samples
 361 parent solution(s) from \mathcal{P}_{t-1} via probabilistic se-
 362 lection (Eq. 2), retrieves relevant experiences $\mathcal{E}_t^{\text{ret}}$
 363 from global memory \mathcal{M}^{glb} , and composes them
 364 with local memory $\mathcal{M}_{t-1}^{\text{loc}}$ as context. Conditioned
 365 on this context, the agent applies controlled mu-
 366 tation (Eq. 3) or compositional crossover (Eq. 4)
 367 to generate an offspring, evaluates it by $\mathcal{F}(\cdot, x)$,
 368 and updates \mathcal{P}_t and $\mathcal{M}_t^{\text{loc}}$ using the reward change
 369 Δ_t (Eq. 5). Finally, CSE returns the best solution
 370 y^* and updates \mathcal{M}^{glb} by distilling intra-task expe-
 371 riences g_τ from top- K improving and degrading
 372 steps (Eq. 6).

373 5 Experiments

374 5.1 Experimental Setup

375 **Benchmark.** We evaluate on EffiBench-X (Qing
 376 et al., 2025), which aggregates 623 algorithmic
 377 problems from major competitive programming
 378 platforms, including AtCoder, Codeforces, and
 379 LeetCode, spanning six programming languages.
 380 Each problem comes with strict time/memory lim-
 381 its and comprehensive test cases. We run exper-
 382 iments on two different languages (Python and
 383 C++) to assess the generality of our method. Fol-
 384 lowing prior work (Qing et al., 2025), we report
 385 three normalized efficiency metrics: Execution-
 386 Time ratio (ET), Memory-Peak ratio (MP) and
 387 Memory-Integral ratio (MI), which compare the
 388 LLM-generated solution with the human reference
 389 on every problem. Detailed definitions and deriva-
 390 tions are provided in Appendix B.

391 **Baselines.** We compare CSE against three code-
 392 generation paradigms: (1) Direct: a single-turn
 393 generation. (2) Self-Reflection: iterative modifi-
 394 cation of the current best solution based on its feed-
 395 back (e.g., variants of EffiLearner (Huang et al.,
 396 2024) and AfterBurner (Du et al., 2025)). (3) SE-
 397 Agent (Lin et al., 2025): a trajectory-level self-
 398 evolving agent method (4) AlphaEvolve (Novikov
 399 et al., 2025): a representative self-evolving agent
 400 method; we reproduce it using the open-source im-
 401 plementation of OpenEvolve (Sharma, 2025). We

	Python			C++			Avg		
	ET	MP	MI	ET	MP	MI	ET	MP	MI
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%	47.47%	48.00%	44.75%	47.98%	48.91%	47.82%
SE-Agent	48.57%	50.39%	49.63%	50.71%	48.25%	47.19%	49.64%	49.32%	48.41%
AlphaEvolve	48.70%	50.33%	51.47%	48.75%	48.67%	46.58%	48.73%	49.50%	49.03%
CSE (Ours)	49.06%	50.99%	52.38%	48.70%	49.03%	47.95%	48.88%	50.01%	50.17%
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%	60.20%	61.18%	45.43%	46.53%	43.16%	51.39%	53.37%	52.17%
SE-Agent	60.09%	59.68%	60.69%	47.74%	46.42%	44.54%	53.92%	53.05%	52.62%
AlphaEvolve	58.86%	60.04%	60.24%	46.19%	46.41%	44.02%	52.53%	53.23%	52.13%
CSE (Ours)	61.06%	61.03%	62.80%	48.75%	46.89%	47.37%	54.91%	53.96%	55.09%
Claude-4.5-Sonnet									
Direct	65.43%	69.03%	67.31%	63.16%	69.16%	62.78%	64.30%	69.10%	65.05%
Self-Reflection	69.79%	71.16%	73.08%	70.29%	68.42%	67.43%	70.04%	69.79%	70.26%
SE-Agent	64.14%	70.58%	67.25%	72.93%	68.81%	70.61%	68.54%	69.70%	68.93%
AlphaEvolve	69.45%	69.50%	70.57%	71.07%	68.45%	67.66%	70.26%	68.98%	69.12%
CSE (Ours)	71.17%	70.39%	73.94%	75.29%	69.82%	74.88%	73.23%	70.11%	74.41%
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%	61.79%	62.08%	56.76%	62.00%	62.59%	60.24%
SE-Agent	61.74%	64.51%	65.36%	68.61%	62.70%	63.46%	65.18%	63.61%	64.41%
AlphaEvolve	63.30%	63.96%	65.53%	64.13%	63.05%	60.54%	63.72%	63.51%	63.04%
CSE (Ours)	65.46%	66.82%	68.10%	69.94%	64.42%	66.83%	67.70%	65.62%	67.47%

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 (higher is better). Avg is the per-metric mean across Python and C++. Best results are in bold.

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 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

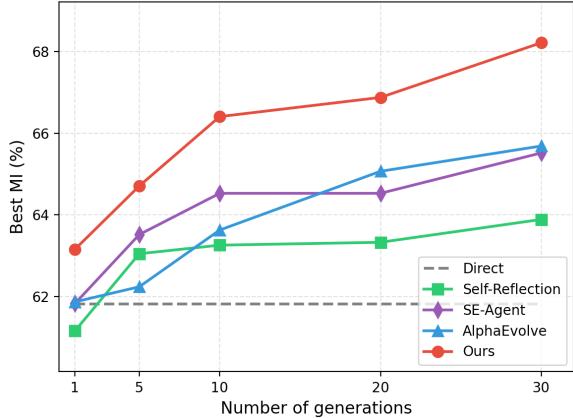


Figure 3: **Best-so-far MI vs. generations.** At each generation t , we report the *best-so-far* MI.

most three 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-

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

Table 2: **Ablation study.** *Planning* means diversified planning initialization; *Evolution* means genetic evolution; *Memory* means hierarchical evolution memory.

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 framework is model-agnostic and captures fundamental principles of efficient code evolution. Compared to population-based baselines AlphaEvolve and SE-Agent, CSE demonstrates clear superiority across most settings. This can be attributed to our key design that enables controlled evolution rather than uncontrolled stochastic exploration: through controlled mutation and compositional crossover, CSE achieves fine-grained, feedback-guided evolution that systematically navigate toward optimal solutions with higher exploration efficiency, while hierarchical memory allows the framework to leverage accumulated experiences from both previous iterations and related tasks to accelerate convergence and avoid repeated failures.

Beyond the final metrics, Figure 3 visualizes how performance evolves with increasing iteration budget. CSE not only reaches a higher best-so-far MI, but also improves more rapidly in early generations and continues to make progress in later generations, suggesting stronger budget utilization under the same 30-generation constraint.

5.3 Further Analysis

Ablation Studies on CSE. To investigate the individual contribution of each component in CSE, we conduct an ablation study where we systematically remove one component at a time while maintaining the same 30-candidate budget. Table 2 presents the results. We evaluate three variants: CSE without *Planning*, which eliminates the diversified initialization strategy; CSE without *Evolution*, which removes the genetic operators; and CSE without *Memory*, which discards the hierarchical evolution memory mechanism. All three components prove essential, with *Memory* showing the largest impact, followed by *Planning* and *Evolution*. The consistent performance drops across both ET and MP demonstrate that CSE’s effectiveness



Figure 4: **Synergy between *Memory* and other modules.** We report MI and the gain ΔMI from adding *Memory* to each module combination.

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 3: **Statistics of Evolution dynamics.**

arises from the synergistic interplay of all three components rather than any single mechanism. We additionally provide qualitative case studies in Appendix D to visualize the evolution dynamics under each ablation.

Analysis of *Memory*. To determine whether *Memory*’s contribution is additive or conditional on other components, we compare different module combinations under the same 30-iteration budget. Fig 4 reports the MI and the gain from adding *Memory* (ΔMI). The results reveal that *Memory*’s effectiveness depends strongly on context. Adding *Memory* to *Planning* alone provides negligible benefit ($\Delta\text{MI} = -0.23$), while combining it with *Evolution* yields substantial gains ($\Delta\text{MI} = +2.68$). The strongest effect emerges when all three components are present ($\Delta\text{MI} = +5.02$). This demonstrates that *Memory* amplifies controlled evolutionary processes rather than providing universal improvement. It is most valuable when the system can generate diverse candidates, refine them through evolution, and strategically revisit solutions—reinforcing guided exploration rather than offering standalone benefits.

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 3 reports

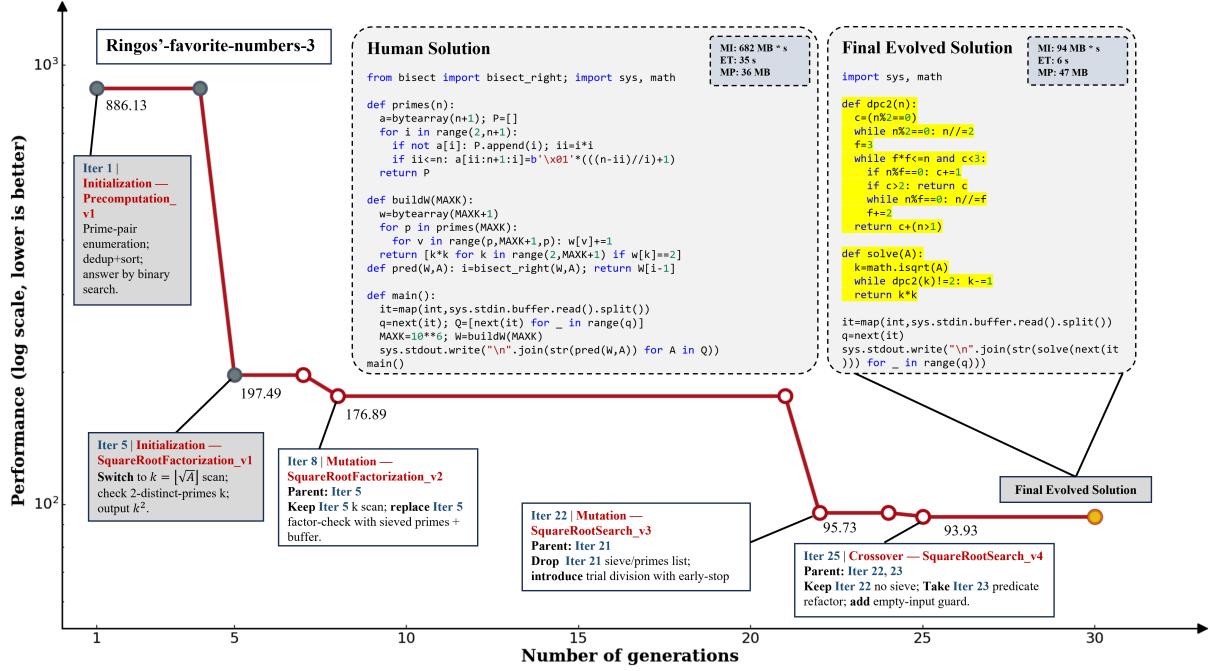


Figure 5: **A Case Study of CSE evolution dynamics.** To quantify progress, we plot the best-so-far **raw memory-time integral** (lower is better) against the number of generations. The case highlights the concrete logic of key initialization, mutation, and crossover steps, and contrasts the final evolved solution against the human solution.

three statistics: (i) the number of improvement times (#Imp.), measuring how many iterations yield efficiency gains over the previous best; (ii) the iteration 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 indicates that CSE continues exploring throughout the entire budget rather than plateauing early, contributing to its superior performance in Table 1.

5.4 Case Study

To demonstrate CSE in practice, Figure 5 shows the complete 30-generation 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, mutations fix bottle-

necks (e.g., inefficient checking or missing guards) while preserving the effective skeleton, yielding steady improvements. 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.

6 Conclusion

We present Controlled Self-Evolution (CSE), a novel framework that addresses the low exploration efficiency of existing self-evolution methods for code optimization. By identifying three fundamental limitations, we propose CSE, including Diversified Planning Initialization, Genetic Evolution and Hierarchical Evolution Memory. Experiments on EffiBench-X demonstrate that CSE consistently outperforms all baselines across various LLM backbones, achieving higher efficiency from early generations while maintaining continuous improvement throughout evolution. Our work highlights the importance of controlled, feedback-driven exploration in self-evolution paradigms for code generation.

500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515

527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552

553 Limitations

554 CSE can continuously improve solution quality
555 through multi-round evolution. However, we have
556 not yet explored how to amortize this iterative op-
557 timization into the base model. A promising yet
558 underexplored direction is to distill CSE’s evolu-
559 tion trajectories into RL-style training (Yu et al.,
560 2025; Zheng et al., 2025; Jiang et al., 2025) signals
561 to strengthen the base model, enabling comparable
562 or better optimization and producing higher-quality
563 solutions.

564 Ethical Considerations

565 This work studies efficiency improvements for
566 algorithmic code generation under standard
567 competitive-programming style evaluation on pub-
568 lic benchmarks. Our experiments do not involve
569 human subjects, user interaction, or personal/sen-
570 sitive data, and are conducted solely on publicly
571 available problem sets and test harnesses. As with
572 prior LLM-based code-generation research, there
573 is a general risk that improved code-generation or
574 optimization techniques could be misused to accel-
575 erate the development of undesirable software. We
576 therefore scope our contribution to research eval-
577 uation and encourage responsible use with appro-
578 priate access controls and adherence to applicable
579 policies.

580 References

- 581 Josh Achiam, Steven Adler, Sandhini Agarwal, Lama
582 Ahmad, Ilge Akkaya, Florencia Leoni Aleman,
583 Diogo Almeida, Janko Altenschmidt, Sam Altman,
584 Shyamal Anadkat, et al. 2023. Gpt-4 technical report.
585 *arXiv preprint arXiv:2303.08774*.
- 586 Zheng Cai, Maosong Cao, Haojong Chen, Kai Chen,
587 Keyu Chen, Xin Chen, Xun Chen, Zehui Chen, Zhi
588 Chen, Pei Chu, et al. 2024. Internlm2 technical re-
589 port. *arXiv preprint arXiv:2403.17297*.
- 590 Mark Chen. 2021. Evaluating large language models
591 trained on code. *arXiv preprint arXiv:2107.03374*.
- 592 Gheorghe Comanici, Eric Bieber, Mike Schaeckermann,
593 Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Mar-
594 cel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al.
595 2025. Gemini 2.5: Pushing the frontier with ad-
596 vanced reasoning, multimodality, long context, and
597 next generation agentic capabilities. *arXiv preprint*
598 *arXiv:2507.06261*.
- 599 Yihong Dong, Xue Jiang, Jiaru Qian, Tian Wang, Kechi
600 Zhang, Zhi Jin, and Ge Li. 2025. A survey on code
601 generation with llm-based agents. *arXiv preprint*
602 *arXiv:2508.00083*.
- 603 Mingzhe Du, Luu Anh Tuan, Yue Liu, Yuhao Qing,
604 Dong Huang, Xinyi He, Qian Liu, Zejun Ma, and
605 See-kiong Ng. 2025. Afterburner: Reinforcement
606 learning facilitates self-improving code efficiency
607 optimization. *arXiv preprint arXiv:2505.23387*.
- 608 Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey,
609 Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman,
610 Akhil Mathur, Alan Schelten, Amy Yang, Angela
611 Fan, et al. 2024. The llama 3 herd of models. *arXiv*
612 *preprint arXiv:2407.21783*.
- 613 Jinyuan Fang, Yanwen Peng, Xi Zhang, Yingxu Wang,
614 Xinhao Yi, Guibin Zhang, Yi Xu, Bin Wu, Siwei
615 Liu, Zihao Li, et al. 2025. A comprehensive survey
616 of self-evolving ai agents: A new paradigm bridg-
617 ing foundation models and lifelong agentic systems.
618 *arXiv preprint arXiv:2508.07407*.
- 619 Huan-ang Gao, Jiayi Geng, Wenyue Hua, Mengkang
620 Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jia-
621 hao Qiu, Xuan Qi, Yiran Wu, et al. 2025. A survey
622 of self-evolving agents: On path to artificial super
623 intelligence. *arXiv preprint arXiv:2507.21046*.
- 624 Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song,
625 Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma,
626 Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: In-
627 centivizing reasoning capability in llms via reinforce-
628 ment learning. *arXiv preprint arXiv:2501.12948*.
- 629 Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie,
630 Kai Dong, Wentao Zhang, Guanting Chen, Xiao
631 Bi, Yu Wu, YK Li, et al. 2024. Deepseek-coder:
632 When the large language model meets programming—
633 the rise of code intelligence. *arXiv preprint*
634 *arXiv:2401.14196*.
- 635 Dong Huang, Jianbo Dai, Han Weng, Puzhen Wu,
636 Yuhao Qing, Heming Cui, Zhijiang Guo, and Jie
637 Zhang. 2024. Effilearner: Enhancing efficiency of
638 generated code via self-optimization. *Advances in
639 Neural Information Processing Systems*, 37:84482–
640 84522.
- 641 Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Day-
642 iheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang,
643 Bowen Yu, Keming Lu, et al. 2024. Qwen2. 5-coder
644 technical report. *arXiv preprint arXiv:2409.12186*.
- 645 Aaron Hurst, Adam Lerer, Adam P Goucher, Adam
646 Perelman, Aditya Ramesh, Aidan Clark, AJ Ost-
647 row, Akila Welihinda, Alan Hayes, Alec Radford,
648 et al. 2024. Gpt-4o system card. *arXiv preprint*
649 *arXiv:2410.21276*.
- 650 Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim,
651 and Sunghun Kim. 2024. A survey on large lan-
652 guage models for code generation. *arXiv preprint*
653 *arXiv:2406.00515*.
- 654 Xue Jiang, Yihong Dong, Mengyang Liu, Hongyi Deng,
655 Tian Wang, Yongding Tao, Rongyu Cao, Binhua
656 Li, Zhi Jin, Wenpin Jiao, et al. 2025. Coderl+:
657 Improving code generation via reinforcement with
658 execution semantics alignment. *arXiv preprint*
659 *arXiv:2510.18471*.

660	Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. 2025. Shinkaevolve: Towards open-ended and sample-efficient program evolution. <i>arXiv preprint arXiv:2509.19349</i> .	714
661		715
662		716
663		717
664	Boaz Lavon, Shahar Katz, and Lior Wolf. 2025. Execution guided line-by-line code generation. <i>arXiv preprint arXiv:2506.10948</i> .	718
665		719
666		
667	Jiaye Lin, Yifu Guo, Yuzhen Han, Sen Hu, Ziyi Ni, Licheng Wang, Mingguang Chen, Hongzhang Liu, Ronghao Chen, Yangfan He, et al. 2025. Se-agent: Self-evolution trajectory optimization in multi-step reasoning with llm-based agents. <i>arXiv preprint arXiv:2508.02085</i> .	720
668		721
669		722
670		723
671		724
672		725
673	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. <i>arXiv preprint arXiv:2412.19437</i> .	726
674		727
675		728
676		
677		
678	Fei Liu, Xialiang Tong, 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 <i>International Conference on Machine Learning (ICML)</i> .	729
679		730
680		731
681		732
682		733
683		
684	Jiawei Liu, Nirav Diwan, Zhe Wang, Haoyu Zhai, Xiaona Zhou, Kiet A Nguyen, Tianjiao Yu, Muntasir Wahed, Yinlin Deng, Hadjer Benkraouda, et al. 2025. Purpcode: Reasoning for safer code generation. <i>arXiv preprint arXiv:2507.19060</i> .	734
685		735
686		736
687		737
688		738
689	Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. Reacc: A retrieval-augmented code completion framework. <i>arXiv preprint arXiv:2203.07722</i> .	739
690		740
691		741
692		742
693	Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Dixin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. <i>arXiv preprint arXiv:2306.08568</i> .	743
694		
695		
696		
697		
698	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. <i>arXiv preprint arXiv:2508.07434</i> .	744
699		745
700		746
701		747
702	Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2023. Self-refine: Iterative refinement with self-feedback. <i>Advances in Neural Information Processing Systems</i> , 36:46534–46594.	748
703		
704		
705		
706		
707		
708	Alexander Novikov, Ngan Vu, 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. <i>arXiv preprint arXiv:2506.13131</i> .	749
709		750
710		751
711		752
712		
713		
714	Yuhao Qing, Boyu Zhu, Mingzhe Du, Zhijiang Guo, Terry Yue Zhuo, Qianru Zhang, Jie M Zhang, Heming Cui, Siu-Ming Yiu, Dong Huang, et al. 2025. Effibench-x: A multi-language benchmark for measuring efficiency of llm-generated code. <i>arXiv preprint arXiv:2505.13004</i> .	753
715		754
716		755
717		756
718		757
719		
720	Bernardino Romera-Paredes, Mohammadamin Barekatain, 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. <i>Nature</i> .	758
721		759
722		760
723		761
724		762
725		763
726		764
727	Asankhaya Sharma. 2025. Opennevolve: an open-source evolutionary coding agent.	765
728		
729	Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. <i>Advances in Neural Information Processing Systems</i> , 36:8634–8652.	766
730		767
731		768
732		769
733		770
734	Haotian Sun, Yuchen Zhuang, Lingkai Kong, Bo Dai, and Chao Zhang. 2023. Adaplaner: Adaptive planning from feedback with language models. <i>Advances in neural information processing systems</i> , 36:58202–58245.	771
735		772
736		773
737		774
738		775
739	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. <i>arXiv preprint arXiv:2501.12599</i> .	776
740		777
741		778
742		779
743		780
744	Evan Wang, Federico Cassano, Catherine Wu, Yunfeng Bai, Will Song, Vaskar Nath, Ziwen Han, Sean Hendryx, Summer Yue, and Hugh Zhang. 2024. Planning in natural language improves llm search for code generation. <i>arXiv preprint arXiv:2409.03733</i> .	781
745		782
746		783
747		784
748		785
749	Huanting Wang, Jingzhi Gong, Huawei Zhang, Jie Xu, and Zheng Wang. 2025a. Ai agentic programming: A survey of techniques, challenges, and opportunities. <i>arXiv preprint arXiv:2508.11126</i> .	786
750		787
751		788
752		789
753	Zhengren Wang, Rui Ling, Chufan Wang, Yongan Yu, Sizhe Wang, Zhiyu Li, Feiyu Xiong, and Wentao Zhang. 2025b. Maintainercoder: Maintainable code generation under dynamic requirements. <i>arXiv preprint arXiv:2503.24260</i> .	790
754		791
755		792
756		793
757		794
758	Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024. Magicoder: empowering code generation with oss-instruct. In <i>Proceedings of the 41st International Conference on Machine Learning</i> , pages 52632–52657.	795
759		796
760		797
761		798
762		799
763	An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengan Huang, Chenxu Lv, et al. 2025. Qwen3 technical report. <i>arXiv preprint arXiv:2505.09388</i> .	800
764		801
765		802
766		803
767		804
768		805
769		806
770		807

767 Feng Yao, Zilong Wang, Liyuan Liu, Junxia Cui,
768 Li Zhong, Xiaohan Fu, Haohui Mai, Vish Krishnan,
769 Jianfeng Gao, and Jingbo Shang. 2025. Training lan-
770 guage models to generate quality code with program
771 analysis feedback. *arXiv preprint arXiv:2505.22704*.

772 Qiyi Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan,
773 Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan,
774 Gaohong Liu, Lingjun Liu, et al. 2025. Dapo: An
775 open-source llm reinforcement learning system at
776 scale. *arXiv preprint arXiv:2503.14476*.

777 Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui
778 Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong
779 Liu, Rui Men, An Yang, et al. 2025. Group
780 sequence policy optimization. *arXiv preprint*
781 *arXiv:2507.18071*.

782 Derui Zhu, Dingfan Chen, Jens Grossklags, Walter
783 Pretschner, Weiyi Shang, et al. More than just func-
784 tional: Llm-as-a-critique for efficient code generation.
785 In *The Thirty-ninth Annual Conference on Neural In-*
786 *formation 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 by instantiating strategy sketches \mathcal{Z} ; (ii) an iterative evolutionary loop performs reward-based parent selection, retrieves inter-task experiences from \mathcal{M}^{glb} and injects them together with the intra-task memory $\mathcal{M}_{t-1}^{\text{loc}}$ into the context, then applies controlled mutation (REFINE) or compositional crossover (CROSSOVER) to generate $y_{\text{child}}^{(t)}$; (iii) after T iterations, we return the best solution y^* and distill task-level experience g_τ from top- K improving/degrading steps to update \mathcal{M}^{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

$$s_i^T = \begin{cases} 0, & \text{if solution fails,} \\ \text{clip}\left(\frac{T_i^H}{T_i^L}, 0, 5\right), & \text{otherwise,} \end{cases} \quad (8)$$

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

$$\text{ET}(\%) = \left(\frac{1}{N} \sum_{i=1}^N s_i^T \right) \times 100\%. \quad (9)$$

(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

$$s_i^M = \begin{cases} 0, & \text{if solution fails,} \\ \text{clip}\left(\frac{M_i^H}{M_i^L}, 0, 5\right), & \text{otherwise,} \end{cases} \quad (10)$$

and the overall MP is

$$\text{MP}(\%) = \left(\frac{1}{N} \sum_{i=1}^N s_i^M \right) \times 100\%. \quad (11)$$

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

$$A = \int_0^{T_{\text{total}}} m(t) dt, \quad (12)$$

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

$$s_i^A = \begin{cases} 0, & \text{if solution fails,} \\ \text{clip}\left(\frac{A_i^H}{A_i^L}, 0, 5\right), & \text{otherwise,} \end{cases} \quad (13)$$

and the overall MI is

$$\text{MI}(\%) = \left(\frac{1}{N} \sum_{i=1}^N s_i^A \right) \times 100\%. \quad (14)$$

C Implementation Details

Experimental Setup Rationale. We follow the evaluation protocol used in EffiLearner (Huang

Algorithm 1 Controlled Self-Evolution (CSE)

Require: Problem specification x , Global memory \mathcal{M}^{glb} , Max iterations T , retrieval queries N_q , top- K size K

Ensure: Optimized solution y^*

- 1: $\mathcal{Z} \leftarrow \text{PlanStrategies}(x)$
- 2: $\mathcal{P}_0 \leftarrow \{y_i^{(0)} \sim \mathcal{A}_\theta(y | x, z_i) \mid z_i \in \mathcal{Z}\}$
- 3: $\mathcal{M}_0^{\text{loc}} \leftarrow \emptyset$
- 4: $\mathcal{S}_\tau \leftarrow \emptyset$
- 5: **for** $t = 1$ **to** T **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: $\text{Op}_t \leftarrow \text{ChooseOp}(\text{Refine}, \text{Crossover})$
- 12: **if** $\text{Op}_t = \text{Refine}$ **then**
- 13: $y_{\text{child}}^{(t)} \leftarrow \text{Refine}(y_{\text{parent}}^{(t)} \mid \text{Context}'_t)$
- 14: **else**
- 15: Sample $y'^{(t)}_{\text{parent}} \leftarrow \text{SelectParent}(\mathcal{P}_{t-1})$ with $P_{\text{select}}(\cdot)$
- 16: $y_{\text{child}}^{(t)} \leftarrow \text{Crossover}(y_{\text{parent}}^{(t)}, y'^{(t)}_{\text{parent}} \mid \text{Context}'_t)$
- 17: **end if**
- 18: $\Delta_t \leftarrow \mathcal{F}(y_{\text{child}}^{(t)}, x) - \mathcal{F}(y_{\text{parent}}^{(t)}, x)$
- 19: $\mathcal{P}_t \leftarrow \text{UpdatePopulation}(\mathcal{P}_{t-1}, y_{\text{child}}^{(t)}, \mathcal{F}(y_{\text{child}}^{(t)}, x))$
- 20: $m_t \leftarrow \text{Reflect}(y_{\text{child}}^{(t)}, y_{\text{parent}}^{(t)}, \Delta_t)$
- 21: $\mathcal{M}_t^{\text{loc}} \leftarrow \text{Compress}(\mathcal{M}_{t-1}^{\text{loc}} \cup \{m_t\})$
- 22: $\mathcal{S}_\tau \leftarrow \mathcal{S}_\tau \cup \{(y_{\text{parent}}^{(t)}, y_{\text{child}}^{(t)}, \Delta_t)\}$
- 23: **end for**
- 24: $y^* \leftarrow \arg \max_{y \in \bigcup_{t=0}^T \mathcal{P}_t} \mathcal{F}(y, x)$
- 25: $\mathcal{S}_\tau^+ \leftarrow \text{TopK}_{\max \Delta}(\mathcal{S}_\tau, K)$
- 26: $\mathcal{S}_\tau^- \leftarrow \text{TopK}_{\min \Delta}(\mathcal{S}_\tau, K)$
- 27: $g_\tau \leftarrow \text{Reflect}(\mathcal{S}_\tau^+, \mathcal{S}_\tau^-, y^*)$
- 28: $\mathcal{M}^{\text{glb}} \leftarrow \mathcal{M}^{\text{glb}} \cup \{g_\tau\}$
- 29: **return** y^*

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.

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 measure-

891
892
893
894
895
896
ment 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.

897
898
899
Reward and Parent Selection. Since smaller
 A indicates better efficiency, we convert A into a
maximization-form reward for selection:

$$900 \quad \mathcal{F} = \frac{1}{A + \epsilon},$$

901
902
where we set $\epsilon = 0.001$. We sample parents pro-
portional to their reward, i.e.,

$$903 \quad p_i = \frac{\mathcal{F}_i}{\sum_j \mathcal{F}_j}.$$

904
905
Failed candidates are assigned $\mathcal{F} = 0$, and thus
will not be selected as parents.

906
907
908
909
910
911
912
913
914
915
916
Search Budget and Operator Schedule. We run
evolution for $T = 30$ iterations. At each iteration,
we generate exactly **one** child candidate, resulting
in a total budget of 30 evolved candidates per task.
We adopt a **deterministic alternating schedule** for
the two evolution operators: **controlled mutation**
and **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.

917
918
919
920
921
922
Initialization and Retrieval Hyperparameters.
We use $N_{\text{init}} = 5$ diverse initialization plans per
problem. For global memory retrieval, we gen-
erate $N_q = 3$ queries and retrieve top- $K_m = 3$
memories per query, resulting in at most $N_q \cdot K_m$
retrieved entries.

923
924
925
926
927
Embedding Model and Similarity Search. We
encode both queries and memory entries using
Qwen3-8B-Embedding. We perform nearest-
neighbor retrieval via **cosine similarity** and return
the top- K_m entries for each query.

928
929
930
931
932
933
934
935
Module Decomposition Template. To enable
controlled mutation and crossover, we enforce a
fixed decomposition template (in Figure 9) and re-
quire the LLM to output code in structured modules
(e.g., [A] Input, [B] Core, [C] Optimization
module). The number of modules is fixed and de-
termined by the template. All subsequent muta-
tion/crossover operations are applied at the module

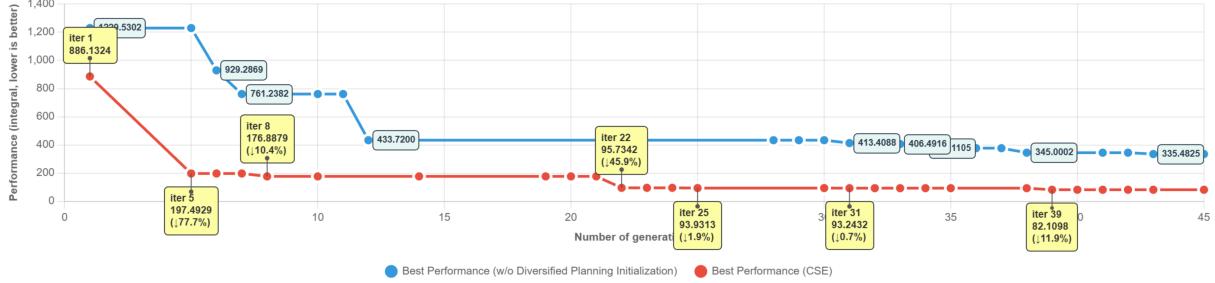
936
937
938
level, which helps maintain interface consistency
and reduces unintended edits outside the targeted
components.

939
940
941
942
943
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**.

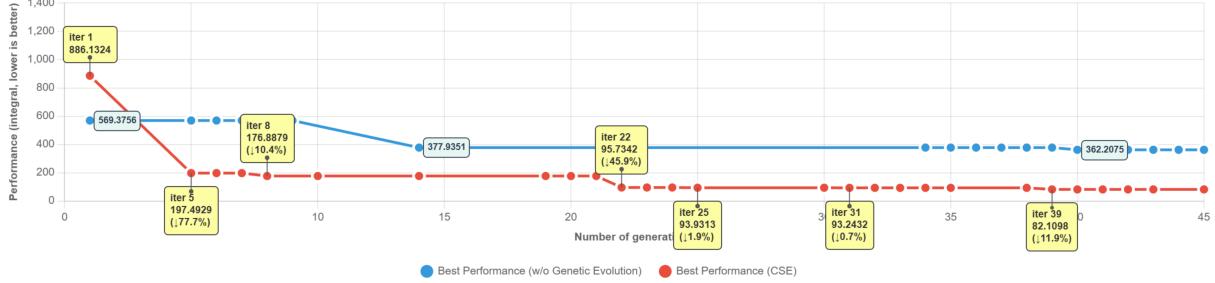
944
945
946
947
948
949
950
951
Global Memory Distillation and Construction
Order. For each task τ , we keep the top- K im-
proving and top- K degrading steps (ranked by
 Δ) to distill a task-level experience item g_τ , with
 $K = 5$. To ensure determinism and reproducibility,
we fix a single task-processing 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 com-
pact experience items and add them into the global
memory for retrieval in subsequent tasks.

D Ablation Case Studies

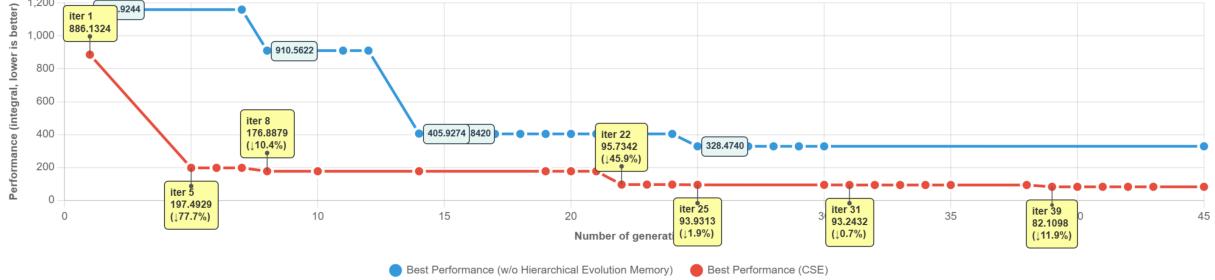
955
956
957
958
959
960
961
962
963
964
Case Study: Diversified Planning Initialization.
Figure 6a compares full CSE with an ablated vari-
ant that removes diversified planning initialization.
We observe a clear initialization bias: without di-
versified planning, the search starts from a sub-
stantially 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 lit-
tle 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), fol-
lowed by continued refinement (iter 8: 176.9).
More importantly, starting from diverse, seman-
tically distinct strategy sketches allows the evo-
lutionary loop to explore multiple high-potential
basins in parallel, which later supports a second
breakthrough (iter 22: 95.7) and a final improve-
ment (iter 25: 93.9). Overall, removing diversified
planning causes prolonged stagnation and a much
higher final plateau (≈ 433.7 vs. 93.9), demon-
strating that diversified planning is crucial for overcom-
ing poor initial modes and accelerating the discov-



(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 Genetic Evolution.** Best-so-far performance over generations for full CSE (red) versus removing genetic 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).

985 ery of high-efficiency solution structures.

986 **Case Study: Genetic Evolution.** Figure 6b illustrates the effect of removing *Genetic Evolution* (i.e., 987 controlled mutation and compositional crossover). 988 Although the ablated variant starts from a relatively 989 strong initial point (best-so-far ≈ 569.4), it quickly 990 becomes stagnant: the curve stays nearly flat for the 991 first ~ 10 generations and only achieves a modest 992 improvement to ≈ 377.9 around the mid-run, after 993 which it plateaus again until the end of the budget. 994 In contrast, full CSE exhibits step-wise breakthroughs 995 characteristic of effective evolutionary search: despite 996 a worse starting best (iter 1: 886.1), it rapidly triggers 997 a major early jump to iter 5 (197.5; $\downarrow 77.7\%$), further refines to iter 8 (176.9), 998 and later achieves another large improvement at 999 1000

1001 iter 22 (95.7; $\downarrow 45.9\%$) followed by a final gain at 1002 iter 25 (93.9). This gap suggests that the core benefit 1003 of Genetic Evolution is not merely producing 1004 more variants, but enabling *structure-preserving, 1005 feedback-aligned* updates: controlled mutation can 1006 surgically repair the bottleneck component without 1007 disrupting already-correct parts (Eq. 3), while 1008 compositional crossover can recombine complementary 1009 strengths across candidates rather than performing 1010 superficial text-level mixing (Eq. 4). Without these 1011 operators, the search lacks an effective mechanism 1012 to translate evaluation feedback into high-reward 1013 structural edits, leading to long plateaus and a much 1014 higher final performance floor (≈ 377.9 vs. 93.9).

1015 **Case Study: Hierarchical Evolution Memory.** Figure 6c examines the impact of removing *Hier-* 1016

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. 5), 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. 6). 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 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; evolution prompts for controlled mutation (reflection and refine) and 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.

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.

1065

User Prompt

Problem Description:
{problem_text}

Required Strategy Count: {k}

1066

Figure 7: 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

1067

* **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}

1068

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.

1069

Figure 8: 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**: 3-5 slots (io_parsing, core_logic, edge_case, perf_patch, misc)
- Diagnose**: Identify bottlenecks, risks, good parts to inherit

Output JSON

```

```json
{
 "solution_name": "String",
 "approach_summary": "String",
 "slot_view": {
 "slots": [{"slot_id": "io_parsing|core_logic|edge_case|perf_patch|misc", "description": "String", "tags": ["impl_method"], "code_span": [start, end]}],
 "diagnoses": [{"slot_id": "String", "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": [...]}]
 }
}
```

```

1070

```

}

```
Rules

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

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

```

1071

## 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).

```

1072

Figure 9: 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

```

1073

```

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}

```

1074

## 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.

1075

Figure 10: Prompts for Controlled Mutation.

## System Prompt

You are an expert competitive programmer specializing in algorithm optimization.

## Principles

- \*\*Correctness First\*\*: Ensure correctness before optimizing for {optimization\_target}.
- \*\*Algorithmic Focus\*\*: Prefer algorithmic improvements over micro-optimizations. Explore directions \*\*different from existing approaches\*\*.
- \*\*Learn from Failures\*\*: Use Memory to prune failed directions, but don’t restrict creative exploration.

1076

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.
```

1078

Figure 11: Prompts for 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."}
  ]
}
```

```

1079

## User Prompt

```
Source Solutions
{source_solutions}

Current Solution
{current_solution}

Current Directions (Strategy Board)
{directions}
```

1080

Figure 12: 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."}
  ]
}
```

```

1081

## User Prompt

```
Source Solutions
{source_solutions}

Current Solution (Failed)
{current_solution}

Current Directions (Strategy Board)
{directions}
```

1082

Figure 13: 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.

1083

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."}
]
}```
```

1084

## User Prompt

```
Directions (Strategy Board)
{directions}
Experiences
{experience_library}
```

1085

Figure 14: 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.

1086

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>"]
}
```

```

1087

## User Prompt

```
Problem Description
{problem_description}

Optimization Target
{optimization_target}

Language
{language}

Local Memory
{local_memory}
```

1088

Figure 15: 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."
}
```

```

1089

```
1090
"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}
```

1091

Figure 16: Prompts for Global Memory Extract.