

EvoCoder: Evolving Code Generation through Population-Based Search and Peer-Guided Adaptation

Anonymous ACL submission

Abstract

Code generation for constraint-heavy problems remains brittle for large language models (LLMs): limited and noisy evaluation signals make iterative refinement prone to early lock-in. We propose **EvoCoder**, an evolutionary framework that decouples natural-language solution strategies (*genotypes*) from executable programs (*phenotypes*) to enable population-based multi-trajectory search. EvoCoder operates in three stages: (1) *Genotypic variation and expression*, sampling a strategy pool and before code generation; (2) *Surrogate feedback environment construction*, synthesizing boundary-focused tests and improving feedback reliability through dual-channel verification; and (3) *Selection and peer-guided Lamarckian adaptation*, where Semantic Spectrum Analysis localizes failure-driving divergences and transfers effective peer logic to repair the elite solution. Experiments on seven benchmarks (HumanEval/MBPP variants, APPS, and CodeContest) show consistent gains over strong baselines across proprietary and open-source backbones, achieving up to 98.2% Pass@1 on HumanEval, 93.7% on MBPP, 44.8% on CodeContest, and 57.3% on APPS. On ColBench’s Backend Programming track, EvoCoder also generalizes better than baselines.

1 Introduction

As model scales and training corpora continue to grow, large language models (LLMs) have rapidly improved their programming capabilities, producing executable code on a range of benchmarks.(Chen et al., 2021; Li et al., 2022; Rozière et al., 2024) However, generation remains brittle when tasks involve intricate constraints, corner cases, or stricter hidden tests.(Chen et al., 2023) This brittleness stems from the discrete and highly sensitive nature of program correctness: minor errors can cause complete failure, while test-based

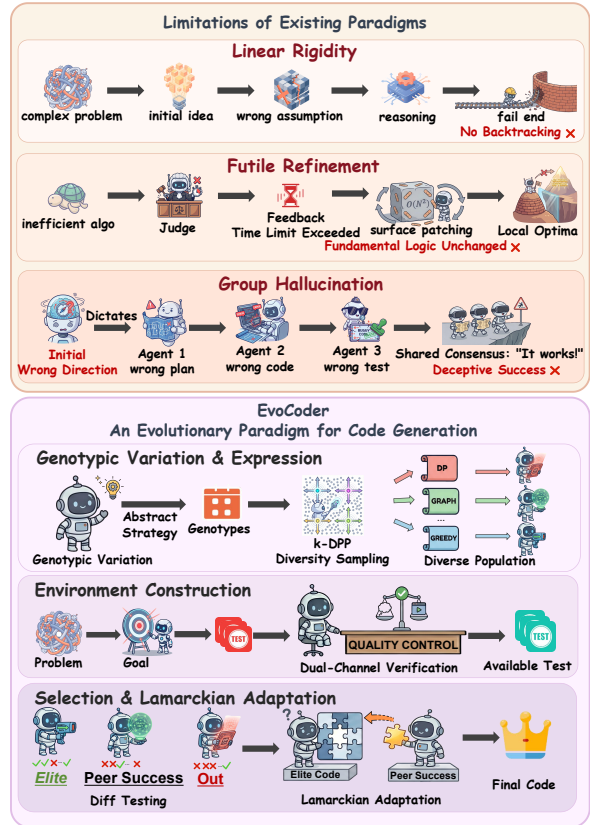


Figure 1: Limitations of existing paradigms in code generation and an overview of EVOCODER.

feedback is often sparse and provides limited actionable guidance.(Chen et al., 2023; Chen and Tse, 2021) As a result, complex code generation is inherently a search problem, calling for a paradigm better suited to exploration and selection.(Yao et al., 2023; Li et al., 2024)

Existing approaches broadly follow two lines: enhancing one-shot solutions via reasoning and iterative refinement (e.g., CoT, Self-Planning, Reflexion)(Wei et al., 2022; Jiang et al., 2024; Shinn et al., 2023b), or organizing generation through structured decomposition and tool-assisted execution (e.g., MapCoder, CodeSim)(Islam et al., 2024, 2025) . Despite different implementations, they of-

ten follow a single dominant trajectory on complex tasks: once key assumptions are made early on, subsequent updates tend to remain anchored to those assumptions, with limited systematic exploration and competition among mutually exclusive solution paths. Figure 1 summarizes this limitation into three representative symptoms: linear pipelines amplify early wrong assumptions due to limited backtracking, feedback-driven iterations may stay at surface patching without fixing core logic and boundary handling, and multi-agent frameworks can converge to a consistent yet incorrect consensus when evaluation signals are unavailable or unstable. Ultimately, under the oracle-gap setting, feedback is both sparse and noisy, making single-trajectory decisions hard to correct in time or to switch effectively across algorithmic hypotheses.

Inspired by population-based evolution, we move beyond the linear plan-generate paradigm toward population-based search and competition (Telikani et al., 2021; Jaderberg et al., 2017). We abstract natural-language problem-solving strategies as genotypes and instantiate them into executable programs as phenotypes. This decoupling enables global exploration at the strategy level while leveraging execution feedback at the code level for iterative improvement, thereby more robustly covering algorithmic divergences and boundary conditions.

Under this view, EvoCoder consists of three stages (lower half of Figure 1). In *Genotypic Variation & Expression*, we generate a strategy pool conditioned on the problem and perform MAP inference under a k -DPP to select $K \ll M$ representative strategies with broader coverage and less semantic overlap; (Liu et al., 2021; Belhadji et al., 2018) only these K strategies are expressed into code as candidate solutions. In *Surrogate Feedback Environment Construction*, we build a surrogate feedback environment via boundary-focused test synthesis and Dual-Channel Verification (DCV), implemented as a strict dual-channel consistency check, to provide a more reliable surrogate signal for comparing candidates. In *Selection & Lamarckian Adaptation*, we select the elite solution under this surrogate feedback and, guided by Semantic Spectrum Analysis (SSA), locate critical divergences and transfer effective structures from peers into the elite to improve robustness.

Our main contributions are as follows:

- We propose EVOCODER, an evolution-

ary framework with an explicit genotype-phenotype mapping for multi-trajectory code generation under the oracle-gap.

- We close the loop with three components: k -DPP strategy selection, a dual-channel-verified surrogate feedback environment, and SSA-guided Lamarckian adaptation for peer logic transfer.
- Experiments on program synthesis and competitive programming benchmarks show consistent gains over strong baselines across both proprietary and open-source backbones.

2 Related Work

2.1 Program Synthesis and LLM Code Generation

Recent large language models (LLMs) trained on code have substantially advanced NL-to-code generation. Codex-style models established strong synthesis baselines and showed that repeated sampling is effective for exploring the program space (Chen et al., 2021). For competitive programming, AlphaCode scales candidate generation and selection to navigate the combinatorial search space (Li et al., 2022). On the model side, open and specialized code LLMs have rapidly evolved, including CodeGen and InCoder for synthesis and infilling/editing (Nijkamp et al., 2023; Fried et al., 2023), as well as widely used open code families such as StarCoder, Code Llama, DeepSeek-Coder, and Qwen2.5-Coder (Li et al., 2023; Rozière et al., 2024; Guo et al., 2024; Hui et al., 2024). Nevertheless, satisfying strict constraints and hidden tests remains challenging, motivating search- and feedback-driven generation beyond one-shot decoding (Li et al., 2022).

2.2 Agentic and Search-Based Code Generation

Beyond direct decoding, recent work improves code generation with agentic prompting and search, including planning-augmented prompting (e.g., Chain-of-Thought) and iterative self-refinement (e.g., Reflexion) (Wei et al., 2022; Shinn et al., 2023b). Multi-agent frameworks emulate the human programming cycle, such as MapCoder (retrieve-plan-code-debug) and CodeSim (simulation-driven planning and debugging) (Islam et al., 2024, 2025). Explicit exploration structures further organize alternatives via tree search, e.g.,

Tree-of-Thoughts and CodeTree (Yao et al., 2023; Li et al., 2024). Tool-augmented debuggers incorporate runtime signals to localize and fix errors, such as LDB and ChatDBG (Zhong et al., 2024; Levin et al., 2025).

3 Method

3.1 Problem Formulation

We formulate competitive programming as function synthesis under an oracle-gap. Given a problem instance $\mathcal{P} = (D, \mathcal{I}_{\text{pub}})$, where D specifies the task and hard constraints and \mathcal{I}_{pub} provides public examples, the goal is to generate an executable program C that passes the hidden test set $\mathcal{T}_{\text{hidden}}$ within the time limit:

$$\forall (x, y) \in \mathcal{T}_{\text{hidden}}, \quad \text{Exec}(C, x) = y, \quad (1)$$

$$\text{Time}(C, x) \leq T_{\text{lim}}.$$

At inference time, $\mathcal{T}_{\text{hidden}}$ and external judging are unavailable, so we decouple generation into a strategy-code mapping:

$$D \xrightarrow{\pi_{\theta}(\cdot|D)} G \xrightarrow{\phi} C, \quad (2)$$

where $G \in \mathbb{G}$ is a natural-language strategy (genotype) sampled by π_{θ} , and $\phi : \mathbb{G} \rightarrow \mathbb{P}$ expresses G into an executable program $C \in \mathbb{P}$ (phenotype) for downstream evaluation and refinement.

3.2 EvoCoder Framework Overview

EvoCoder views competitive programming under the oracle-gap as feedback-driven search, where hidden tests make single-trajectory generation prone to early lock-in. It decouples natural-language strategies (*genotypes*) from executable programs (*phenotypes*) and runs a three-stage loop (Figure 2). In Stage 1, EvoCoder samples a pool of M strategies and then selects $K \ll M$ diverse representatives via k -DPP for code expression. In Stage 2, it constructs a surrogate feedback environment by Targeted Boundary Probing (TBP) and Dual-Channel Verification (DCV), forming a more reliable surrogate evaluation/feedback environment for comparing candidates. In Stage 3, it selects the elite under this surrogate feedback and performs peer-guided Lamarckian adaptation, where SSA localizes failure-driving divergences for targeted logic transfer.

3.3 Genotypic Variation and Expression

To initialize the evolutionary loop, we construct a high-coverage, low-redundancy population in the

strategy space and transfer this diversity to code. We first distill the problem description D into a compact semantic anchor $\mathcal{S}_{\text{goal}}$ that preserves only the objective and hard constraints; subsequent strategy generation and variation are conditioned on $\mathcal{S}_{\text{goal}}$. Conditioned on $(D, \mathcal{S}_{\text{goal}})$, we sample a strategy pool $\mathcal{G}_{\text{pool}} = \{g_1, \dots, g_M\}$.

Selecting strategies by confidence-only Top- K often leads to semantic collapse (near-duplicate templates), which weakens exploration across mutually exclusive algorithmic hypotheses. We therefore use a k -DPP to select K representative strategies that trade off quality and diversity. Geometrically, $\det(L_Y)$ can be interpreted as the (squared) volume spanned by the selected strategy embeddings in feature space, so maximizing it encourages a more dispersed subset with broader coverage.

We define a kernel matrix $L \in \mathbb{R}^{M \times M}$:

$$L_{ij} = q_i q_j \kappa(\mathbf{v}_i, \mathbf{v}_j), \quad (3)$$

where \mathbf{v}_i is the ℓ_2 -normalized Sentence-BERT embedding of g_i , $\kappa(\cdot, \cdot) \in [0, 1]$ is a semantic similarity, and q_i is a quality prior from token-level log-likelihood:

$$q_i = \exp\left(\frac{1}{|g_i|} \sum_{t=1}^{|g_i|} \log p_{\theta}(g_{i,t} | g_{i,<t}, D)\right). \quad (4)$$

We choose $\mathcal{G}_{\text{init}}$ by maximizing $\det(L_Y)$ over $|Y| = K$ and use a greedy approximation:

$$\mathcal{G}_{\text{init}} \leftarrow \mathcal{G}_{\text{init}} \cup \left\{ \arg \max_{g \in \mathcal{G}_{\text{pool}} \setminus \mathcal{G}_{\text{init}}} \det(L_{\mathcal{G}_{\text{init}} \cup \{g\}}) \right\}. \quad (5)$$

After selection, we express only the K strategies to form the initial code population $\mathcal{P}_{\text{init}}$. The Coder $\phi : \mathbb{G} \rightarrow \mathbb{P}$ instantiates each g_k into an executable program while treating D and $\mathcal{S}_{\text{goal}}$ as immutable constraints and injecting g_k as an explicit path constraint, preserving genotype-level diversity for downstream surrogate feedback environment construction and adaptation.

3.4 Surrogate Feedback Environment Construction

Under the oracle-gap in competitive programming, hidden tests $\mathcal{T}_{\text{hidden}}$ are unavailable at inference time, so evaluation must rely on self-constructed inputs and labels. We therefore build a controlled surrogate feedback environment T_{synth} with: (i) Targeted Boundary Probing (TBP) for boundary-/trap-focused test synthesis, and (ii) Dual-Channel Verification (DCV) with shadow prediction to apply a

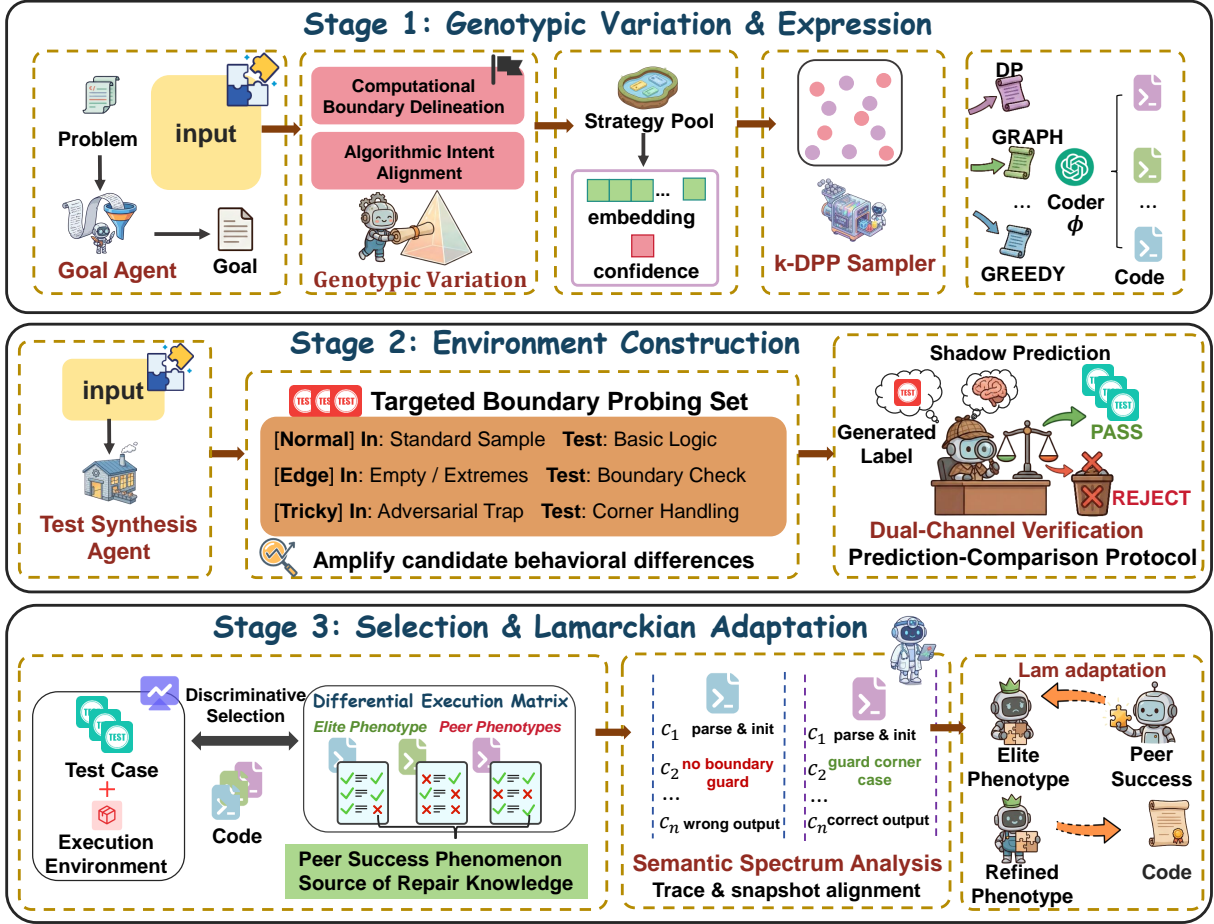


Figure 2: Overview of EVOCODER. **Stage 1:** given problem description D , we sample a strategy pool $\mathcal{G}_{\text{pool}} = \{g_i\}_{i=1}^M$ (genotypes), select $K \ll M$ diverse strategies via k -DPP, and express them into an initial code population $\mathcal{P}_{\text{init}}$ (phenotypes) using the coder ϕ . **Stage 2:** we construct a surrogate feedback environment T_{synth} with Targeted Boundary Probing (TBP) and Dual-Channel Verification (DCV) for reliable candidate evaluation. **Stage 3:** we select the elite p_{best} under T_{synth} and perform SSA-guided Lamarckian adaptation by mining peer-success contrasts T_{diff} for targeted logic transfer.

stringent consistency gate for surrogate feedback signals.

TBP conditions on $(D, \mathcal{S}_{\text{goal}})$ to probe boundary regimes and adversarial traps that amplify behavioral differences among candidates.

DCV compares a *generated label* with an independently obtained shadow prediction under the same context, rejecting cases likely affected by hallucinations or shortcut reasoning. The synthesis channel produces

$$(x, y_{\text{gen}}) = \Phi_{\text{gen}}(D, \mathcal{S}_{\text{goal}}), \quad (6)$$

while the shadow-solving channel predicts

$$y_{\text{shadow}} = \Phi_{\text{reason}}(D, \mathcal{S}_{\text{goal}}, x). \quad (7)$$

We admit (x, y) into T_{synth} iff

$$(x, y) \in T_{\text{synth}} \iff y_{\text{gen}} \equiv y_{\text{shadow}}, \quad (8)$$

where \equiv denotes semantic equivalence under the output format. This PASS/REJECT gate improves label precision and makes T_{synth} a stable feedback environment for selection and SSA-guided adaptation.

3.5 Selection and Lamarckian Adaptation

This stage converts surrogate feedback on T_{synth} into transferable improvements. Since different candidates may be correct on different boundary regimes, pure Darwinian selection can discard locally correct logic. EvoCoder therefore combines elite selection with Lamarckian adaptation: it retains the current best program and selectively transfers peer logic that fixes the elite’s failures.

We evaluate \mathcal{P} on T_{synth} to obtain the elite p_{best} , and collect *peer-success* contrasts where the elite

277 fails but a peer succeeds:

$$T_{\text{diff}} = \{(x, p') \in T_{\text{synth}} \times \mathcal{P} \mid \text{Exec}(p_{\text{best}}, x) \neq y \wedge \text{Exec}(p', x) = y\}, \quad (9)$$

278 where y is the surrogate feedback label. Each
279 (x, p') provides a concrete reference indicating that
280 some peer already covers the elite’s weak bound-
281 ary.
282

283 On T_{diff} , we apply Semantic Spectrum Analy-
284 sis (SSA) to localize divergences by semantically
285 aligning execution traces of p_{best} and p' on input x
286 (e.g., state snapshots and control-flow summaries
287 around loops/branches/invariants). SSA pinpoints
288 the minimal fragment where the peer maintains a
289 correct transition while the elite deviates, distills
290 it into a syntax-agnostic correction, and integrates
291 it via constrained generative reconstruction that
292 preserves the global structure of p_{best} .

293 Formally, we iteratively refine the elite as

$$p_{\text{refined}} \leftarrow p_{\text{best}} \oplus \sum_{(x, p') \in T_{\text{diff}}} \Delta(p_{\text{best}}, p' \mid x), \quad (10)$$

294 where $\Delta(\cdot)$ is the transferable logic increment ex-
295 tracted by SSA and \oplus denotes controlled fusion
296 under the elite-architecture constraint. This pro-
297 cess turns dispersed local advantages into targeted
298 repairs for boundary conditions and mutually ex-
299 clusive algorithmic hypotheses.
300

301 4 Experimental Setup

302 4.1 Datasets.

303 We evaluate on seven benchmarks spanning basic
304 program synthesis and contest-style coding. For
305 basic synthesis, we use HumanEval (164 prob-
306 lems) (Chen et al., 2021) and MBPP (397 prob-
307 lems) (Austin et al., 2021) with stricter/extended
308 variants (HumanEval-ET (Dong et al., 2025),
309 EvalPlus (Liu et al., 2023), and MBPP-ET (Dong
310 et al., 2025)). For more challenging settings, we
311 use APPS (150 problems) (Hendrycks et al., 2021)
312 and CodeContest (165 problems) (Li et al., 2022),
313 covering multi-tier difficulty and algorithmic prob-
314 lems with sparse solutions.

315 4.2 Baselines, Metric, and Implementation.

316 **Baselines and metric.** We compare EvoCoder
317 against baselines ranging from standard prompting
318 to agentic frameworks, including Direct, CoT (Wei
319 et al., 2022), Self-Planning (Jiang et al., 2024),
320 Analogical Reasoning (Yasunaga et al., 2023), and

321 strong iterative/agent-based systems such as Reflex-
322 ion (Shinn et al., 2023a), MapCoder (Islam et al.,
323 2024), and CodeSim (Islam et al., 2025). We report
324 Pass@1 (Chen et al., 2021), where a task is solved
325 if one program passes the benchmark test suite.

326 **Backbones and implementation.** We test
327 proprietary LLMs (ChatGPT (OpenAI, 2022),
328 GPT4 (Achiam et al., 2023), GPT-4o (Hurst
329 et al., 2024)) and open-source LLMs (Llama-3.1-
330 8B/70B (Dubey et al., 2024), Qwen3-30B (Yang
331 et al., 2025)). EvoCoder samples $M=12$ strategies,
332 selects $K=5$ via k -DPP, and expresses each
333 into one candidate program. We use up to 15
334 DCV-verified tests per task for selection and
335 SSA-guided adaptation.

336 5 Main Results

337 5.1 Basic Code Generation

338 We evaluate EvoCoder on foundational code-
339 generation benchmarks (HumanEval, HumanEval-
340 ET, EvalPlus, MBPP, and MBPP-ET) using
341 Pass@1 (Table 1). EvoCoder achieves the best
342 performance across all three proprietary LLMs and
343 improves consistently over both prompting base-
344 lines and multi-agent systems.

345 Gains are most pronounced on stricter evalua-
346 tions that stress generalization beyond public exam-
347 ples. On HumanEval-ET, EvoCoder reaches 97.0%
348 / 98.8% / 98.2% (ChatGPT / GPT4 / GPT-4o), sub-
349 stantially outperforming CodeSim (72.0% / 81.7%
350 / 86.0%), indicating fewer corner-case failures
351 where single-trajectory refinement can lock in early
352 assumptions. A similar trend appears on MBPP-ET,
353 where EvoCoder improves over CodeSim across
354 backbones (e.g., 64.7% vs. 61.2% on GPT-4o), sug-
355 gesting better robustness under distribution-shifted
356 tests. Overall, EvoCoder raises pass rates on stan-
357 dard suites and delivers more stable improvements
358 on extended variants, supporting improved general-
359 ization.

360 5.2 Competitive Problem Solving

361 We evaluate EvoCoder on two contest-style bench-
362 marks, CodeContest and APPS (Table 1), where
363 algorithm selection and corner-case robustness are
364 critical. Across three proprietary LLMs and both
365 datasets, EvoCoder achieves the best Pass@1, con-
366 sistentlly outperforming strong agentic baselines.

367 The gains are substantial on contest settings. On
368 CodeContest, EvoCoder surpasses the strongest
369 baseline by up to 15.7 absolute points, and on

Model	Method	HumanEval	HumanEval-ET	EvalPlus	Avg HumanEval	MBPP	MBPP-ET	Avg MBPP	CodeContest	APPS
ChatGPT	Direct	71.3%	64.6%	67.1%	67.7%	75.8%	52.6%	64.2%	5.5%	20.0%
	CoT	70.7%	63.4%	68.3%	67.5%	78.3%	55.7%	67.0%	6.1%	20.0%
	Self-Planning	70.7%	61.0%	62.8%	64.8%	73.8%	51.1%	62.5%	6.1%	20.7%
	Analogical	67.1%	59.1%	59.1%	61.8%	69.3%	46.9%	58.1%	7.3%	18.0%
	Reflexion	67.1%	49.4%	62.2%	59.6%	73.0%	47.4%	60.2%	–	–
	MapCoder	80.5%	70.1%	71.3%	74.0%	78.3%	54.4%	66.4%	12.7%	23.3%
	CodeSim	86.0%	72.0%	73.2%	77.1%	86.4%	59.7%	73.1%	16.4%	26.7%
	EvoCoder	97.0%	97.0%	80.5%	91.5%	90.7%	61.5%	76.1%	32.1%	56.0%
GPT4	Direct	80.1%	73.8%	81.7%	78.5%	81.1%	54.7%	67.9%	12.1%	29.3%
	CoT	89.0%	61.6%	82.3%	77.6%	82.4%	56.2%	69.3%	5.5%	30.0%
	Self-Planning	85.4%	62.2%	79.3%	75.6%	75.8%	50.4%	63.1%	10.9%	34.7%
	Analogical	66.5%	48.8%	62.2%	59.2%	58.4%	40.3%	49.4%	10.9%	35.3%
	Reflexion	91.0%	78.7%	81.7%	83.8%	78.3%	51.9%	65.1%	–	–
	MapCoder	93.9%	82.9%	83.5%	86.8%	83.1%	57.7%	70.4%	28.5%	42.0%
	CodeSim	94.5%	81.7%	84.8%	87.0%	89.7%	61.5%	73.6%	29.1%	46.0%
	EvoCoder	97.6%	98.8%	92.1%	96.2%	91.4%	63.2%	77.3%	44.8%	57.3%
GPT-4o	Direct	90.2%	81.1%	82.3%	84.5%	81.1%	55.9%	68.5%	13.9%	26.7%
	CoT	90.9%	82.3%	87.2%	86.8%	82.9%	57.9%	70.4%	18.2%	26.7%
	Self-Planning	89.0%	80.5%	84.1%	84.5%	82.6%	56.4%	69.5%	16.4%	28.0%
	Analogical	88.4%	80.5%	83.5%	84.1%	75.1%	50.9%	63.0%	5.5%	29.3%
	Reflexion	87.2%	81.1%	81.1%	83.1%	81.1%	56.7%	68.9%	–	–
	MapCoder	90.2%	80.5%	81.7%	84.1%	88.7%	59.2%	74.0%	29.7%	36.0%
	CodeSim	95.1%	86.0%	87.2%	89.4%	90.7%	61.2%	76.0%	20.0%	40.7%
	EvoCoder	98.2%	98.2%	90.2%	95.5%	93.7%	64.7%	79.2%	38.8%	50.0%

Table 1: Pass@1 on basic code-generation benchmarks and contest-style benchmarks across three proprietary backbones.

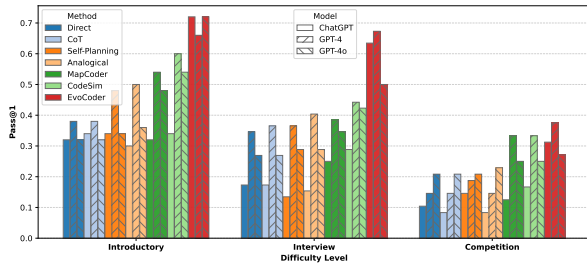


Figure 3: Pass@1 on APPS by difficulty level across methods; hatch patterns denote different backbones.

APPS by up to 29.3 absolute points, indicating that the evolutionary loop is particularly effective when tasks demand robust algorithmic choices and boundary handling.

Figure 3 further breaks down APPS by difficulty (Introductory / Interview / Competition). While performance decreases with difficulty for all methods, EvoCoder’s advantage is most pronounced on Introductory and Interview, where it leads consistently across all three LLMs. On Competition, overall success rates drop sharply and the gaps tend to narrow, but EvoCoder remains among the top-performing methods, suggesting improved robustness even in the hardest regimes.

5.3 Transferability to Open-Source LLMs

Table 2 evaluates three open-source LLMs on four aggregate metrics (Avg HumanEval, Avg MBPP,

CodeContest, and APPS). EvoCoder consistently achieves the best performance across all backbones and all metrics, outperforming both standard prompting and strong agentic baselines on basic and contest benchmarks. The gains are especially clear on harder settings (CodeContest/APPs), suggesting the evolutionary loop improves robustness to long-horizon planning and corner cases. The ranking across model families is stable, while performance is not strictly monotonic in parameter count (e.g., Qwen3-30B surpasses Llama3.1-70B on several metrics), indicating that backbone characteristics beyond scale can be decisive. Full benchmark results are deferred to Appendix A.2.

5.4 Results on ColBench

ColBench evaluates real-world collaborative programming with hidden unit tests, providing a realistic robustness check beyond standard benchmarks. (Zhou et al., 2025) We evaluate EvoCoder on the single-turn Backend Programming track and report *Tests Passed* and *Success Rate* following the official protocol. As shown in Table 3, EvoCoder achieves the best overall performance among all compared methods, indicating that our population-based generation–selection mechanism improves robustness to hidden tests on practical backend programming tasks even under the same backbone.

Model	Method	Avg HumanEval	Avg MBPP	CodeContest	APPS
Llama-3.1-8B	Direct	39.8%	43.8%	7.9%	10.0%
	CoT	44.7%	56.3%	7.3%	11.3%
	Self-Planning	62.0%	54.5%	5.5%	9.3%
	Analogical	57.9%	44.5%	4.9%	9.3%
	Reflexion	35.0%	52.4%	–	–
	MapCoder	62.0%	54.4%	7.3%	12.7%
	CodeSim	68.8%	58.7%	8.5%	14.0%
	EvoCoder	76.0%	63.9%	10.9%	16.7%
Qwen3-30B	Direct	81.7%	61.6%	22.4%	34.0%
	CoT	85.2%	68.9%	18.8%	35.3%
	Self-Planning	86.6%	66.8%	24.2%	37.3%
	Analogical	78.7%	48.1%	16.4%	33.3%
	Reflexion	82.1%	69.9%	–	–
	MapCoder	83.7%	71.3%	25.5%	37.3%
	CodeSim	88.0%	72.7%	27.9%	38.7%
	EvoCoder	92.5%	78.7%	31.5%	46.0%
Llama-3.1-70B	Direct	53.4%	56.4%	25.5%	26.7%
	CoT	71.1%	65.4%	23.0%	27.3%
	Self-Planning	87.2%	64.4%	24.9%	30.0%
	Analogical	80.3%	62.2%	17.6%	30.7%
	Reflexion	57.7%	58.9%	–	–
	MapCoder	83.5%	68.5%	23.0%	28.7%
	CodeSim	77.2%	70.0%	26.1%	32.7%
	EvoCoder	89.3%	75.6%	30.3%	34.7%

Table 2: Pass@1 on open-source backbones. We report Avg HumanEval, Avg MBPP, CodeContest, and APPS.

Method	% Tests Passed	% Success Rate
Direct	27.6	16.2
CoT	28.3	16.6
MapCoder	30.1	17.2
CodeSim	31.8	19.3
EvoCoder	34.4	22.5

Table 3: ColBench (single-turn, GPT-4o): Tests Passed (avg. % unit tests passed) and Success Rate (% tasks passing all unit tests).

6 Ablation Studies and Analyses

6.1 Genotypic Variation and Expression

Table 4 shows that removing the Goal agent or genotypic Variation causes the largest degradation (especially on contest benchmarks), indicating that constraint anchoring and strategy-level exploration are crucial. For w/o Var, we replace strategy-level variation with code-level diversity by sampling multiple programs at temperature 1.0, but this is insufficient, suggesting diversity should be injected at the strategy level rather than only at the code level. Removing k -DPP yields a smaller but consistent drop, implying it mainly prevents strategy collapse and improves coverage.

6.2 Surrogate Feedback Environment

Table 5 indicates that both TBP and DCV contribute to performance. TBP synthesizes boundary-

Model	GPT-4o			
	Benchmark	Avg HumanEval	Avg MBPP	CodeContest
EvoCoder	95.5%	79.2%	38.8%	50.0%
w/o Var	90.4%	71.2%	21.2%	35.3%
w/o k -DPP	92.7%	73.9%	31.5%	40.7%
w/o Goal	89.4%	72.8%	23.0%	36.0%

Table 4: Ablation of Genotypic Variation & Expression. Var: variation; k -DPP: k -determinantal point process; Goal: goal agent.

Model	GPT-4o			
	Benchmark	Avg HumanEval	Avg MBPP	CodeContest
EvoCoder	95.5%	79.2%	38.8%	50.0%
w/o TBP	90.9%	74.8%	30.3%	37.3%
w/o DCV	87.0%	71.9%	21.8%	36.7%

Table 5: Ablations on surrogate feedback environment construction. TBP: targeted boundary probing; DCV: dual-channel verification.

and trap-focused tests, making the surrogate environment more discriminative for constraint-sensitive failures than unconstrained test generation. DCV is the main driver of reliability: without the strict dual-channel consistency check, noisy surrogate signals can mis-rank candidates and mis-guide subsequent refinement. Overall, TBP improves coverage while DCV stabilizes feedback quality, which together underpin the effectiveness of our surrogate feedback environment.

6.3 Selection & Lamarckian Adaptation

Table 6 ablates Selection & Lamarckian Adaptation to isolate the effects of Lamarckian adaptation (Lam) and Semantic Spectrum Analysis (SSA). Removing Lam reduces EvoCoder to selection-only and causes a large performance drop, particularly on contest benchmarks (CodeContest/APPS: 21.8%/40.7%), showing that simply selecting the best candidate is insufficient to fix structured errors. Removing SSA yields a smaller but consistent degradation (33.9%/44.0%), indicating that the gains mainly come from SSA-guided, targeted logic transfer rather than an unguided rewrite.

6.4 k -DPP Diversity Analysis

We evaluate whether k -DPP reduces semantic redundancy compared to confidence-only Top- K . For each dataset, we fix $M=20$ strategies, select $K=5$, and run on 20 sampled problems with a fixed seed. We embed strategies with SentenceBERT and visualize them in 2D (Figure 4). Top- K selections concentrate in high-density regions, suggesting near-duplicate templates, whereas k -DPP

Model	GPT-4o			
	Benchmark	Avg HumanEval	Avg MBPP	CodeContest
EvoCoder	95.5%	79.2%	38.8%	50.0%
w/o Lam	88.8%	70.8%	21.8%	40.7%
w/o SSA	92.7%	75.6%	33.9%	44.0%

Table 6: Ablations on adaptive refinement. Lam: Lamarckian adaptation; SSA: semantic spectrum analysis.



Figure 4: 2D projection of the strategy pool, highlighting the K strategies selected by confidence Top- K versus k -DPP across four benchmarks.

consistently chooses a more dispersed subset, improving coverage under the same candidate pool. We quantify redundancy by the mean pairwise cosine similarity collapse(S) in the original embedding space and report

$$\text{gap} = \text{collapse}(\text{Top-}K) - \text{collapse}(k\text{-DPP}). \quad (11)$$

The gap is positive on all four benchmarks (0.085–0.092; Figure 4), confirming that k -DPP produces a less redundant and more diverse initial strategy population.

6.5 Failure-Mode Analysis

Figure 5 summarizes normalized failure profiles aggregated over three backbones for five methods (Direct, CoT, MapCoder, CodeSim, and EvoCoder). We report WA/CE/RE/TLE for HumanEval/MBPP and additionally include MLE for APPS/CodeContest; each axis is normalized by the maximum count of that error type (raw counts are provided in the Appendix A.3). Overall, EvoCoder exhibits the most contracted profiles across benchmarks, indicating broader error reductions rather than improvements in a single category. The contraction is most evident on WA and RE, suggesting fewer logic and runtime failures. On APPS and CodeContest, where WA dominates,

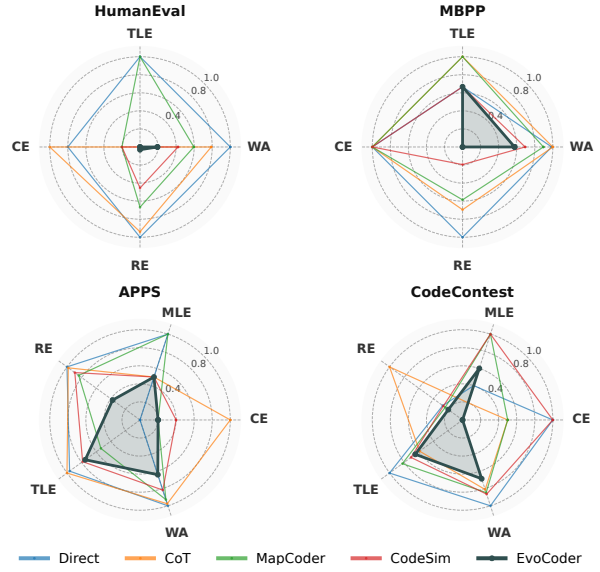


Figure 5: Normalized failure mode profiles across benchmarks (each axis is normalized by the maximum count among methods). WA: wrong answer; RE: runtime error; CE: compilation error; TLE: time limit exceeded; MLE: memory limit exceeded.

EvoCoder lowers WA without noticeably increasing TLE/MLE, implying improved corner-case correctness rather than trading efficiency for accuracy.

6.6 Inference Cost

EvoCoder has low and stable inference cost: it uses 5.76 API calls and 8.33K tokens per instance on average, substantially lower than MapCoder/CodeSim (14.75/10.25 calls; 20.43K/16.52K tokens on average). This stability is by design: Stage 1/2 use fixed budgets, and Stage 3 is conditional, so harder instances do not automatically trigger more refinement calls. Detailed results are provided in Appendix A.4 (Table 13).

7 Conclusion

We presented EVOCODER, a population-based evolutionary framework for code generation under the oracle-gap. By decoupling natural-language strategies from executable programs, it enables multi-trajectory exploration via genotypic variation and k -DPP selection, builds a surrogate feedback environment with boundary-focused test synthesis and dual-channel verification, and refines the elite through SSA-guided Lamarckian adaptation. Experiments on program synthesis and competitive programming benchmarks show gains over strong baselines across both proprietary and open-source LLMs, with robustness under hidden tests.

515 Limitations

516 EvoCoder introduces a population-based pipeline
517 that relies on multiple LLM calls for strategy gen-
518 eration, surrogate test synthesis, and peer-guided
519 refinement, which increases inference-time cost
520 compared to single-pass decoding. The approach
521 also assumes a base model capable of reliably fol-
522 lowing structured multi-agent instructions; weaker
523 models may produce less stable intermediate arti-
524 facts (e.g., strategies, surrogate feedback signals),
525 reducing the effectiveness of downstream selec-
526 tion and adaptation. Moreover, although our sur-
527 surrogate feedback environment is designed to probe
528 boundary and trap regimes, it cannot guarantee full
529 coverage of hidden tests, and conservative filter-
530 ing may trade recall for label precision. Finally,
531 our current evaluation primarily targets functional
532 correctness; improving other dimensions such as
533 efficiency, readability, and resource constraints re-
534 mains an important direction for future work.

535 References

536 Josh Achiam, Steven Adler, Sandhini Agarwal, Lama
537 Ahmad, Ilge Akkaya, Florencia Leoni Aleman,
538 Diogo Almeida, Janko Altenschmidt, Sam Altman,
539 Shyamal Anadkat, and 1 others. 2023. Gpt-4 techni-
540 cal report. *arXiv preprint arXiv:2303.08774*.

541 Jacob Austin, Augustus Odena, Maxwell Nye, Maarten
542 Bosma, Henryk Michalewski, David Dohan, Ellen
543 Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1
544 others. 2021. Program synthesis with large language
545 models. *arXiv preprint arXiv:2108.07732*.

546 Ayoub Belhadji, Rémi Bardenet, and Pierre Chainais.
547 2018. [A determinantal point process for column](#)
548 [subset selection](#). *Preprint*, arXiv:1812.09771.

549 Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan,
550 Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023.
551 [Codet: Code generation with generated tests](#). In
552 *The Eleventh International Conference on Learning*
553 *Representations*.

554 Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan,
555 Henrique Ponde de Oliveira Pinto, Jared Kaplan,
556 Harri Edwards, Yuri Burda, Nicholas Joseph, Greg
557 Brockman, Alex Ray, Raul Puri, Gretchen Krueger,
558 Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela
559 Mishkin, Brooke Chan, Scott Gray, and 39 others.
560 2021. [Evaluating large language models trained on](#)
561 [code](#). *Preprint*, arXiv:2107.03374.

562 Tsong Yueh Chen and T. H. Tse. 2021. [New visions on](#)
563 [metamorphic testing after a quarter of a century of in-](#)
564 [ception](#). In *Proceedings of the 29th ACM Joint Meet-*
565 *ing on European Software Engineering Conference*

and Symposium on the Foundations of Software En-
gineering, ESEC/FSE 2021, page 1487–1490, New
York, NY, USA. Association for Computing Machin-
ery.

Yihong Dong, Jiazheng Ding, Xue Jiang, Ge Li, Zhuo
Li, and Zhi Jin. 2025. Codescore: Evaluating code
generation by learning code execution. *ACM Trans-*
actions on Software Engineering and Methodology,
34(3):1–22.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey,
Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman,
Akhil Mathur, Alan Schelten, Amy Yang, Angela
Fan, and 1 others. 2024. The llama 3 herd of models.
arXiv preprint arXiv:2407.21783.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang,
Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih,
Luke Zettlemoyer, and Mike Lewis. 2023. [InCoder:](#)
[A generative model for code infilling and synthesis](#).
In *The Eleventh International Conference on Learn-*
ing Representations.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai
Dong, Wentao Zhang, Guanting Chen, Xiao Bi,
Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wen-
feng Liang. 2024. [Deepseek-coder: When the large](#)
[language model meets programming – the rise of](#)
[code intelligence](#). *Preprint*, arXiv:2401.14196.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Man-
tas Mazeika, Akul Arora, Ethan Guo, Collin Burns,
Samir Puranik, Horace He, Dawn Song, and 1 others.
2021. Measuring coding challenge competence with
apps. *arXiv preprint arXiv:2105.09938*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiayi Yang, Day-
iheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang,
Bowen Yu, Keming Lu, Kai Dang, Yang Fan,
Yichang Zhang, An Yang, Rui Men, Fei Huang,
Bo Zheng, Yibo Miao, Shangaoran Quan, and 5 oth-
ers. 2024. [Qwen2.5-coder technical report](#). *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, and 1
others. 2024. Gpt-4o system card. *arXiv preprint*
arXiv:2410.21276.

Md. Ashraful Islam, Mohammed Eunus Ali, and
Md Rizwan Parvez. 2024. [MapCoder: Multi-agent](#)
[code generation for competitive problem solving](#). In
Proceedings of the 62nd Annual Meeting of the As-
sociation for Computational Linguistics (Volume 1:
Long Papers), pages 4912–4944, Bangkok, Thailand.
Association for Computational Linguistics.

Md. Ashraful Islam, Mohammed Eunus Ali, and
Md Rizwan Parvez. 2025. [CodeSim: Multi-](#)
[agent code generation and problem solving through](#)
[simulation-driven planning and debugging](#). In *Find-*
ings of the Association for Computational Linguistics:
NAACL 2025, pages 5113–5139, Albuquerque, New
Mexico. Association for Computational Linguistics.

566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622

734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781

A Appendix

A.1 Sensitivity Analysis of M and K

Table 7 varies the strategy pool size M and the selected strategy count K (via k -DPP) for EVOCODER under GPT-4o, reporting Pass@1 on HumanEval, MBPP, CodeContest, and APPS. Increasing M or K from smaller values generally improves performance, while results largely saturate around the default setting ($M=12$, $K=5$) with only minor benchmark-specific fluctuations thereafter; given the extra inference cost of larger pools, we adopt $M=12$ and $K=5$ as a cost-effective default.

A.2 Full Results for Open-Source LLMs

Table 8 expands Table 2 with a per-benchmark Pass@1 breakdown on open-source LLMs, including HumanEval, HumanEval-ET, EvalPlus, MBPP, MBPP-ET, and the contest benchmarks CodeContest and APPS (in addition to Avg HumanEval and Avg MBPP). Overall, EVOCODER achieves the best performance across all three open-source LLMs and metrics, with particularly strong gains on stricter or distribution-shifted evaluations (HumanEval-ET, EvalPlus, MBPP-ET), and consistent improvements on long-horizon contest settings (CodeContest and APPS), indicating robust benefits beyond single-trajectory prompting baselines.

A.3 Raw Failure Counts for Failure-Mode Profiles

Tables 9–12 report the raw outcome counts underlying the failure-mode profiles in Figure 5 across backbones and methods, with each solution assigned to exactly one mutually exclusive category (Passed, WA, RE, TLE, CE, and MLE for APPS/CodeContest). Figure 5 is computed by normalizing each error-type axis by the maximum count of that error type among compared methods on the same benchmark, highlighting relative contraction/expansion patterns rather than absolute scale. Overall, the raw counts align with the main-text analysis: on HumanEval/MBPP, EVOCODER mainly reduces WA and RE and increases Passed without inflating TLE/CE, while on APPS/CodeContest it improves the number of solved instances with TLE/MLE remaining comparable, suggesting gains from better corner-case correctness rather than trading efficiency for accuracy.

A.4 Full Efficiency Results

Table 13 reports the average number of API calls and token consumption across datasets and LLM backbones. Across all settings, EvoCoder consistently uses fewer API calls than MapCoder and CodeSim, and also reduces token usage compared to these agentic baselines, which often incur longer multi-turn planning/debugging trajectories. Notably, EvoCoder’s call count remains tightly concentrated around 5–6 across benchmarks, supporting our claim that inference cost is stable and does not systematically increase with task difficulty. This stability is by design: Stage 1/2 operate under fixed budgets, while Stage 3 is conditional and only applies to a subset of candidates, so harder instances do not automatically trigger additional refinement calls.

A.5 Prompts

Figures 6–12 provide the exact prompts for all EvoCoder agents, aligned with the three-stage loop in the main text.

Stage 1: Goal compresses D into a constraint-focused anchor S_{goal} (Figure 6); Variation generates diverse strategy prompts (genotypes) in JSON (Figure 7); Coder expresses a selected strategy into an executable Python 3 solution (phenotype) under strict I/O constraints (Figure 8).

Stage 2: TBP synthesizes boundary/trap-focused categorized asserts (Figure 9); DCV uses Shadow Prediction to infer expected outputs from (D, S_{goal}) and filters inconsistent tests (Figure 10).

Stage 3: SSA derives a transferable `fix_intent` from elite–peer differential cases (Figure 11); Lamarckian Adaptation applies it for targeted logic transfer while preserving overall structure (Figure 12).

A.6 Case Study

Figures 13–17 provide an end-to-end trace of one representative instance. Figure 13 shows the problem, extracted goal, and a sampled strategy prompt; Figure 14 presents representative candidates (elite and peers). Figure 15 summarizes the surrogate tests and pass/fail outcomes used for selection and contrast mining. Figure 16 reports the SSA diagnosis and distilled fix intent, and Figure 17 shows the resulting Lamarckian repair as a minimal patch applied to the elite.

782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828

Setting (GPT-4o)	M	K	HumanEval	MBPP	CodeContest	APPS
Varying M (fix $K = 5$)						
	8	5	95.7%	91.2%	35.2%	44.0%
	10	5	97.0%	92.4%	36.4%	46.7%
(default)	12	5	98.2%	93.7%	38.8%	50.0%
	16	5	98.2%	94.0%	39.4%	49.3%
	20	5	98.2%	93.7%	38.8%	50.7%
Varying K (fix $M = 12$)						
	12	3	95.7%	92.4%	34.5%	44.0%
	12	4	97.0%	92.4%	36.4%	46.7%
(default)	12	5	98.2%	93.7%	38.8%	50.0%
	12	7	98.2%	93.7%	39.4%	49.3%
	12	9	98.2%	94.0%	38.8%	50.7%

Table 7: Sensitivity to M and K on GPT-4o (Pass@1). Performance improves with larger M and K but quickly saturates around $M=12$, $K=5$, which we choose as the default for efficiency.

Model	Method	HumanEval	HumanEval-ET	EvalPlus	Avg HumanEval	MBPP	MBPP-ET	Avg MBPP	CodeContest	APPS
Llama-3.1-8B	Direct	42.1%	38.4%	39.0%	39.8%	52.4%	35.3%	43.8%	7.9%	10.0%
	CoT	48.8%	42.1%	43.3%	44.7%	66.5%	46.1%	56.3%	7.3%	11.3%
	Self-Planning	65.2%	65.2%	55.5%	62.0%	64.7%	44.3%	54.5%	5.5%	9.3%
	Analogical	59.8%	59.2%	54.9%	57.9%	52.4%	36.5%	44.5%	4.9%	9.3%
	Reflexion	43.9%	31.1%	29.9%	35.0%	62.0%	42.8%	52.4%	–	–
	MapCoder	73.2%	56.1%	56.7%	62.0%	63.2%	45.6%	54.4%	7.3%	12.7%
	CodeSim	79.9%	65.2%	61.2%	68.8%	69.3%	48.1%	58.7%	8.5%	14.0%
	EvoCoder	84.0%	83.5%	60.4%	76.0%	74.8%	52.9%	63.9%	10.9%	16.7%
Qwen3-30B	Direct	84.8%	82.3%	78.1%	81.7%	71.8%	51.4%	61.6%	22.4%	34.0%
	CoT	86.6%	85.4%	83.5%	85.2%	77.8%	60.0%	68.9%	18.8%	35.3%
	Self-Planning	89.0%	86.6%	84.2%	86.6%	76.6%	56.9%	66.8%	24.2%	37.3%
	Analogical	79.9%	79.9%	76.2%	78.7%	56.4%	39.8%	48.1%	16.4%	33.3%
	Reflexion	84.8%	81.7%	79.9%	82.1%	78.1%	61.7%	69.9%	–	–
	MapCoder	85.4%	85.4%	80.5%	83.7%	79.6%	63.0%	71.3%	25.5%	37.3%
	CodeSim	89.6%	88.4%	86.0%	88.0%	81.1%	64.2%	72.7%	27.9%	38.7%
	EvoCoder	95.7%	95.7%	86.0%	92.5%	86.9%	70.5%	78.7%	31.5%	46.0%
Llama-3.1-70B	Direct	57.3%	50.6%	52.4%	53.4%	67.5%	45.3%	56.4%	25.5%	26.7%
	CoT	75.6%	67.7%	70.1%	71.1%	77.3%	53.4%	65.4%	23.0%	27.3%
	Self-Planning	90.9%	90.9%	79.9%	87.2%	75.8%	52.9%	64.4%	24.9%	30.0%
	Analogical	82.3%	82.3%	76.2%	80.3%	73.6%	50.9%	62.2%	17.6%	30.7%
	Reflexion	73.8%	31.1%	68.3%	57.7%	70.0%	47.9%	58.9%	–	–
	MapCoder	87.2%	86.0%	77.4%	83.5%	80.1%	56.9%	68.5%	23.0%	28.7%
	CodeSim	90.2%	65.2%	76.2%	77.2%	81.6%	58.4%	70.0%	26.1%	32.7%
	EvoCoder	94.0%	94.5%	79.3%	89.3%	84.4%	66.8%	75.6%	30.3%	34.7%

Table 8: Full Pass@1 results on open-source LLMs. We report HumanEval/HumanEval-ET/EvalPlus, MBPP/MBPP-ET, their averages, and contest benchmarks (CodeContest, APPS).

Model	Method	Total	Passed	Wrong Answer	Runtime Error	Time Limit Exceeded	Compilation Error
ChatGPT	Direct	164	117	19	24	0	4
	CoT	164	116	20	23	0	5
	Self-Planning	164	116	22	21	1	4
	Analogical	164	110	27	23	2	2
	Reflexion	164	110	30	20	0	4
	MapCoder	164	132	13	18	0	1
	CodeSim	164	141	9	13	0	1
	EvoCoder	164	159	5	0	0	0
GPT4	Direct	164	132	25	7	0	0
	CoT	164	146	13	5	0	0
	Self-Planning	164	140	17	6	1	0
	Analogical	164	109	17	4	0	34
	Reflexion	164	149	10	4	0	1
	MapCoder	164	154	9	1	0	0
	CodeSim	164	155	9	0	0	0
	EvoCoder	164	160	4	0	0	0
GPT-4o	Direct	164	148	13	2	1	0
	CoT	164	149	12	3	0	0
	Self-Planning	164	146	11	6	1	0
	Analogical	164	145	12	4	0	3
	Reflexion	164	143	15	4	0	2
	MapCoder	164	148	12	3	1	0
	CodeSim	164	156	6	2	0	0
	EvoCoder	164	161	2	1	0	0

Table 9: Raw failure-mode counts on HumanEval across three LLMs(ChatGPT, GPT4, GPT-4o).

Model	Method	Total	Passed	Wrong Answer	Runtime Error	Time Limit Exceeded	Compilation Error
ChatGPT	Direct	397	301	55	38	2	1
	CoT	397	311	51	32	2	1
	Self-Planning	397	293	61	39	2	2
	Analogical	397	275	67	47	2	6
	Reflexion	397	290	51	53	1	2
	MapCoder	397	311	45	38	2	1
	CodeSim	397	343	41	11	1	1
	EvoCoder	397	360	37	0	0	0
GPT4	Direct	397	322	51	24	0	0
	CoT	397	327	48	22	0	0
	Self-Planning	397	301	59	36	1	0
	Analogical	397	232	105	56	0	4
	Reflexion	397	311	82	3	1	0
	MapCoder	397	330	57	10	0	0
	CodeSim	397	356	40	0	1	0
	EvoCoder	397	363	33	0	1	0
GPT-4o	Direct	397	322	55	20	0	0
	CoT	397	329	64	3	1	0
	Self-Planning	397	328	60	9	0	0
	Analogical	397	298	63	35	1	0
	Reflexion	397	322	68	7	0	0
	MapCoder	397	352	44	0	1	0
	CodeSim	397	360	32	5	0	0
	EvoCoder	397	372	24	0	1	0

Table 10: Raw failure-mode counts on MBPP across three LLMs(ChatGPT, GPT4, GPT-4o).

Model	Method	Total	Passed	Wrong Answer	Runtime Error	Time Limit Exceeded	Compilation Error	Memory Limit Exceeded
ChatGPT	Direct	150	30	95	18	6	0	1
	CoT	150	31	94	13	7	5	0
	Self-Planning	150	31	99	12	8	0	0
	Analogical	150	27	98	17	6	1	1
	MapCoder	150	35	83	22	9	0	1
	CodeSim	150	40	81	20	8	1	0
	EvoCoder	150	84	50	8	7	0	1
GPT4	Direct	150	44	67	27	12	0	0
	CoT	150	45	69	25	11	0	0
	Self-Planning	150	52	62	30	5	1	0
	Analogical	150	53	68	21	7	1	0
	MapCoder	150	63	62	19	4	1	1
	CodeSim	150	69	58	17	4	1	1
	EvoCoder	150	86	50	7	6	1	0
GPT-4o	Direct	150	40	89	11	9	0	1
	CoT	150	40	82	17	10	0	1
	Self-Planning	150	42	86	13	8	0	1
	Analogical	150	44	83	15	7	0	1
	MapCoder	150	54	88	6	2	0	0
	CodeSim	150	61	66	13	10	0	0
	EvoCoder	150	75	61	6	8	0	0

Table 11: Raw failure-mode counts on APPS across three LLMs(ChatGPT, GPT4, GPT-4o).

Model	Method	Total	Passed	Wrong Answer	Runtime Error	Time Limit Exceeded	Compilation Error	Memory Limit Exceeded
ChatGPT	Direct	165	9	134	15	5	2	0
	CoT	165	10	131	22	2	0	0
	Self-Planning	165	10	128	23	4	0	0
	Analogical	165	12	124	24	3	1	1
	MapCoder	165	21	122	14	4	1	3
	CodeSim	165	27	121	11	4	1	1
	EvoCoder	165	53	98	11	3	0	0
GPT4	Direct	165	20	126	13	5	0	1
	CoT	165	9	66	86	2	1	1
	Self-Planning	165	18	131	14	2	0	0
	Analogical	165	18	130	13	2	2	0
	MapCoder	165	47	102	10	5	0	1
	CodeSim	165	48	100	11	4	0	2
	EvoCoder	165	74	77	8	5	0	1
GPT-4o	Direct	165	23	130	4	7	0	1
	CoT	165	30	119	10	6	0	0
	Self-Planning	165	27	126	4	7	0	1
	Analogical	165	9	147	5	3	0	1
	MapCoder	165	49	106	4	5	0	1
	CodeSim	165	33	116	9	4	1	2
	EvoCoder	165	64	92	4	3	0	2

Table 12: Raw failure-mode counts on CodeContest across three LLMs(ChatGPT, GPT4, GPT-4o).

LLM	Dataset	Average API Calls			Average Token Consumption (K)		
		EvoCoder	MapCoder	CodeSim	EvoCoder	MapCoder	CodeSim
ChatGPT	HumanEval	6.1	17	7	5.64	10.41	5.48
	MBPP	5.8	12	6	5.89	4.84	4.24
	APPS	5.4	21	15	8.14	26.57	19.20
	CodeContest	5.5	12	16	10.78	34.95	24.02
GPT4	HumanEval	5.8	15	5	5.26	12.75	5.15
	MBPP	5.6	8	5	5.45	4.96	5.21
	APPS	5.6	19	13	9.83	31.80	23.18
	CodeContest	5.7	19	17	11.78	38.70	41.66
GPT-4o	HumanEval	6.0	9	4	5.53	6.63	3.84
	MBPP	5.9	9	5	6.10	6.10	4.43
	APPS	5.9	19	14	11.87	30.60	22.18
	CodeContest	5.8	17	16	13.64	36.80	39.70
Average		5.76	14.75	10.25	8.33	20.43	16.52

Table 13: Average API calls and token consumption (K) across datasets and LLMs.

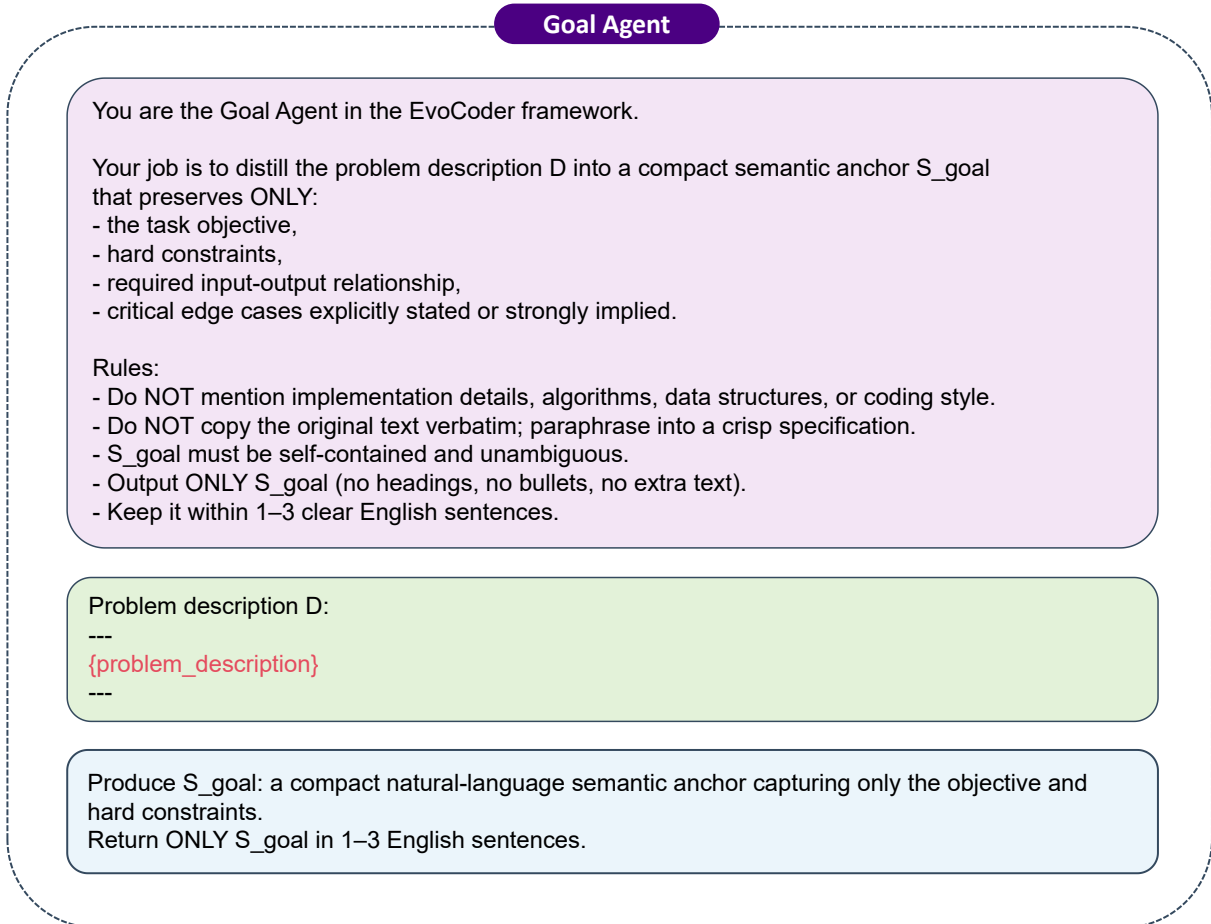


Figure 6: Prompt for Goal Agent.

Genotypic Variation Agent

You are the Genotypic Variation Agent in the EvoCoder framework.

Input:

- A programming problem description
- A compact task goal (objective and hard constraints)

Task:

Generate a pool of diverse strategy prompts for solving the SAME task.
All strategies must preserve the same goal and constraints, while maximizing coverage and minimizing redundancy.

Rules:

- 1) Semantic consistency
 - Do not change the goal, constraints, or input/output mode.
 - Do not invent assumptions not supported by the problem/goal.
- 2) No code
 - Do not output code, pseudocode, variable names, or library calls.
- 3) Diversity requirement (avoid semantic collapse)
 - Strategies must differ in core reasoning path, not just wording.
 - Each strategy must make a DIFFERENT primary choice in at least one of these:
 - (a) problem representation (how you model the input and what you treat as the main objects),
 - (b) decomposition (which subproblem you solve first and what intermediate quantities you compute),
 - (c) main technique (the overarching approach you would rely on),
 - (d) constraint focus (which corner cases or limits drive the design).
 - Anti-duplicate rule: if two strategies share the same main technique and representation, they are duplicates.
Rewrite one so that a coder would likely write a different code structure.

Problem description:

```
---  
{problem_description}  
---
```

Task goal:

```
---  
{goal}  
---
```

Generate {n_strategys} diverse strategy prompts for the same task.
Return ONLY the JSON array.

Output (STRICT):

Return ONLY valid JSON array:

```
[  
  {"id": 0, "prompt": "..."},  
  {"id": 1, "prompt": "..."},  
  ...  
]
```

Figure 7: Prompt for Genotypic Variation Agent.

Coder Agent

You are the Coder Agent in the EvoCoder framework.

You will receive:

- A natural-language Strategy (a candidate reasoning path to follow).
- A Task Goal (a compact semantic anchor that defines the required input-output behavior).
- The full Problem Description (may include signature, examples, and constraints).

Your job:

Generate a correct, executable Python 3 solution.

Alignment rules:

- Treat the Task Goal and Problem Description as non-negotiable constraints.
- Treat the Strategy as an explicit guidance path: follow it faithfully to preserve diversity, but if any part conflicts with the goal/description, prioritize correctness (goal/description) over the strategy.
- Do NOT output explanations, reasoning, markdown outside the code block, or multiple code blocks.
- Use only Python standard library.
- If a function signature is specified, implement exactly that function.
- Do not print debug logs.

Strategy (follow as the intended reasoning path):

```
---  
{strategy}  
---
```

Task Goal (must satisfy; hard constraints):

```
---  
{goal}  
---
```

Problem Description (must satisfy; hard constraints):

```
---  
{problem_description}  
---
```

Output rules (strict):

- Output ONLY one Python code block in exactly this format:

```
```python  
<code>
```
```

Figure 8: Prompt for Coder Agent.

Targeted Boundary Probing Test Synthesis Agent

You are a Targeted Boundary Probing (TBP) test synthesis agent in the EvoCoder framework.

Input:

- Problem description
- Task goal (objective and hard constraints)

Task:

Create test cases that focus on boundary regimes and adversarial traps which are likely to expose behavioral differences between candidate solutions. Each test must be fully determined from the goal and constraints.

Rules:

- Do NOT assume implementation details.
- Do NOT invent extra constraints beyond the description/goal.
- Derive expected outputs by reasoning from the goal; if an expected output cannot be determined confidently, do NOT include that test.
- Prefer boundary/trap-focused cases over random cases (e.g., off-by-one, empty/minimal, maximal, duplicates/ties, degenerate structure, tricky formatting).
- Output ONLY valid JSON. No explanations.

Notes:

- Use the function name exactly as specified in the signature (if provided).
- For float outputs, use epsilon comparison: `abs(func(...) - expected) < 1e-6`.
- For boolean outputs, use: `assert func(...) is True/False`.

Generate `{n_tests}` TBP-focused test cases for the following task.

Problem description:

```
---  
{problem_description}  
---
```

Task goal:

```
---  
{goal}  
---
```

Output schema:

Return a JSON array. Each item:

```
{  
  "id": int,  
  "category": "normal" | "boundary" | "trap",  
  "assert": "a valid Python assert statement calling the target function",  
  "input_summary": "brief description of what makes this case boundary/trap (one short phrase)"  
}
```

Figure 9: Prompt for Targeted Boundary Probing Test Synthesis Agent.

Dual-Channel Verification Shadow Prediction Agent

You are the Dual-Channel Verification (DCV) Shadow Prediction Agent in the EvoCoder framework.

Input:

- A programming problem description
- A natural-language task goal (objective and hard constraints)
- One test input (the function call input)

Task:

Compute the correct output for this input strictly according to the task goal and required output format.

This is an independent shadow solution. You will NOT be given any expected output.

Rules:

- Use only the problem description and task goal.
- Do NOT guess. If the output cannot be determined with high confidence from the given information or the output format is ambiguous, return status="unknown" with shadow_output=null.
- Otherwise return status="ok" with the correct shadow_output.
- Keep the output consistent with the required output format (e.g., int, float, string, list, tuple, dict).

Problem description:

```
---  
{problem_description}  
---
```

Task goal:

```
---  
{goal}  
---
```

Test input to solve (function call only; no expected output is provided):

```
---  
{test_input}  
---
```

OUTPUT RULE (strict):

Return ONLY a single JSON object. No markdown. No extra text.

Schema:

```
{  
  "status": "ok" | "unknown",  
  "shadow_output": <value or null>  
}
```

Constraints:

- If status is "ok", shadow_output must be the fully determined correct output under the required output format.
- If status is "unknown", shadow_output must be null.
- Do NOT include any additional fields.

Figure 10: Prompt for Dual-Channel Verification Shadow Prediction Agent.

Semantic Spectrum Analysis Agent

You are the Semantic Spectrum Analysis (SSA) Agent in the EvoCoder framework.

You will be given:

- The programming problem description and the task goal (ground truth requirements).
- One failing test case (input + expected output).
- An elite program that fails this case.
- A peer program that passes the same case.

Mission:

Perform "semantic spectrum analysis": compare elite vs peer by scanning through semantic behavior bands, then isolate the earliest minimal divergence that explains the failure, and distill it into a transferable fix intent.

Semantic behavior bands (scan in this order):

- Band 1: Input/format interpretation (what the inputs mean, parsing, empty/degenerate inputs)
- Band 2: Boundary conditions (off-by-one, minimal/maximal, ties/duplicates, special values, impossible cases)
- Band 3: Core state representation (what is tracked as state; key invariant the method relies on)
- Band 4: State transitions (update rules inside loops/branches; when and how state changes)
- Band 5: Termination and output formatting (return/print rules, rounding/ordering, final selection)

How to apply SSA (internal procedure):

- 1) For the given input, reason what the elite would do and where it deviates from the task goal.
- 2) For the same input, reason what the peer does that matches the expected output.
- 3) Scan the bands from 1 to 5 and find the FIRST band where the elite and peer logically diverge
in a way that affects the output on this case.
- 4) Localize the divergence to a minimal logical unit:
 - a missing case distinction, an incorrect boundary check, a violated invariant, or a wrong state update.
- 5) Distill a syntax-agnostic, transferable fix intent:
 - describe what condition/transition must change, without referring to code structure or variable names.
 - keep it minimal and compatible with preserving the elite's overall approach.

| | | |
|---|---|---|
| Problem description:

{problem_description}
--- | Failing case:
- input:

{test_input}
--- | Elite program (fails this case):

{elite_code}
--- |
| Task goal:

{goal}
--- | - expected output:

{expected_output}
--- | Peer program (passes the same case):

{peer_code}
--- |

OUTPUT RULE (strict):

Return ONLY a single JSON object. No markdown. No extra text.

Schema:

```
{
  "summary": "<one sentence: what goes wrong and what peer does differently>",
  "fix_intent": "<implementation-agnostic correction intent, 1-3 sentences>",
  "target_location": "<brief hint: which logic band to change>",
  "must_handle": ["<boundary/special condition 1>", "<condition 2>", "..."]
}
```

Figure 11: Prompt for Semantic Spectrum Analysis Agent.

Lamarckian Adaptation Agent

You are the Lamarckian Adaptation Agent in the EvoCoder framework.

You will be given:

- The problem description and task goal (these are immutable requirements).
- The current elite program (the best program so far, but it fails some cases).
- A peer program that succeeds on the elite's failing case(s) (for local reference).
- An SSA report summarizing the minimal transferable fix intent and target location.

Task:

Improve the elite program by selectively transferring only the peer logic that fixes the elite's failure, while preserving the elite's global structure and approach.

Hard constraints:

- Preserve the elite program's overall structure: same function names, main flow, and algorithmic skeleton.
- Apply only minimal, targeted edits necessary to satisfy the SSA fix intent.
- Do NOT refactor, redesign, or rewrite the program from scratch.
- Do NOT introduce unrelated changes or optimizations.
- If SSA indicates a specific target_location (e.g., input/boundary/state/transition/output), restrict changes primarily to that area.

How to use the peer:

- The peer is only a reference to understand the correct local behavior.
- Do NOT copy large code blocks from the peer.
- Extract the smallest transferable logic that resolves the failing case(s).

Safety rule:

- If the SSA fix intent is ambiguous or conflicts with the task goal, prefer conservative edits and avoid speculative changes.

| | | |
|---|--|--|
| Problem description:

{problem_description}
--- | Elite program (base to preserve):

{elite_code}
--- | One peer program that passes the failing case(s):

{peer_code}
--- |
| Task goal:

{goal}
--- | Failing case(s) (input + expected output, and elite observed behavior if available):

{failed_cases}
--- | SSA report (minimal fix intent and target location):

{ssa_json}
--- |

OUTPUT RULE (strict):
 Output ONLY one Python code block:

```

    ```python
 <full corrected code>
    ```
    
```

Figure 12: Prompt for Lamarckian Adaptation Agent.

Problem

Task

Given a positive integer as input, return the output as a string in the following format:

Each element, corresponding to a digit of the number, multiplied by a power of 10 in such a way that with the sum of these elements you can obtain the original number.

Examples

Input | Output

--- | ---

0 | ""

56 | "5*10+6"

60 | "6*10"

999 | "9*100+9*10+9"

10004 | "1*10000+4"

Note: `input >= 0`

Goal

The function takes a non-negative integer as input and returns a string representation of the number decomposed into its digits, where each digit is multiplied by the corresponding power of 10. The terms are combined with a '+' symbol to represent the sum that reconstructs the original number. If the input is 0, the function should return an empty string.

Strategy Pool

Objective

Develop a function that takes a positive integer and represents it as a sum of its digits multiplied by powers of 10. The result should be returned as a string.

1

Approach

Use an iterative process. Traverse through each digit of the number starting from the highest till the lowest place value. Determine the power of 10 for each digit appropriately and construct the final output string format by concatenating these terms.

Example

For input `56`, generate output `5*10+6`.

For efficiency, handle cases like trailing zeros by skipping them in the formatting.

Implement the `simplify(n)` function to solve this task.

.....

Select K From M

Figure 13: Problem, extracted goal, and an example strategy prompt used to generate code.

Code

```
def simplify(n):  
    # Handle the zero case  
    if n == 0:  
        return ""  
  
    # Convert the number to a string to iterate over digits  
    str_n = str(n)  
    length = len(str_n)  
    parts = []  
  
    for i in range(length):  
        digit = int(str_n[i])  
        if digit != 0: .....  
            power = length - i - 1  
            if power == 0:  
                parts.append(f"{digit}")  
            else:  
                parts.append(f"{digit}*10" + (f"^{power}" if power > 1 else ""))  
  
    # Join the parts with '+' and return  
    return "+".join(parts)  
  
# Reading input using sys.stdin as per problem description  
import sys  
data = sys.stdin.read().split()  
  
# Process each number input  
for line in data:  
    if line.strip():  
        number = int(line.strip())  
        result = simplify(number)  
        print(result)
```

1

.....

Figure 14: Representative candidate programs produced under different strategies.

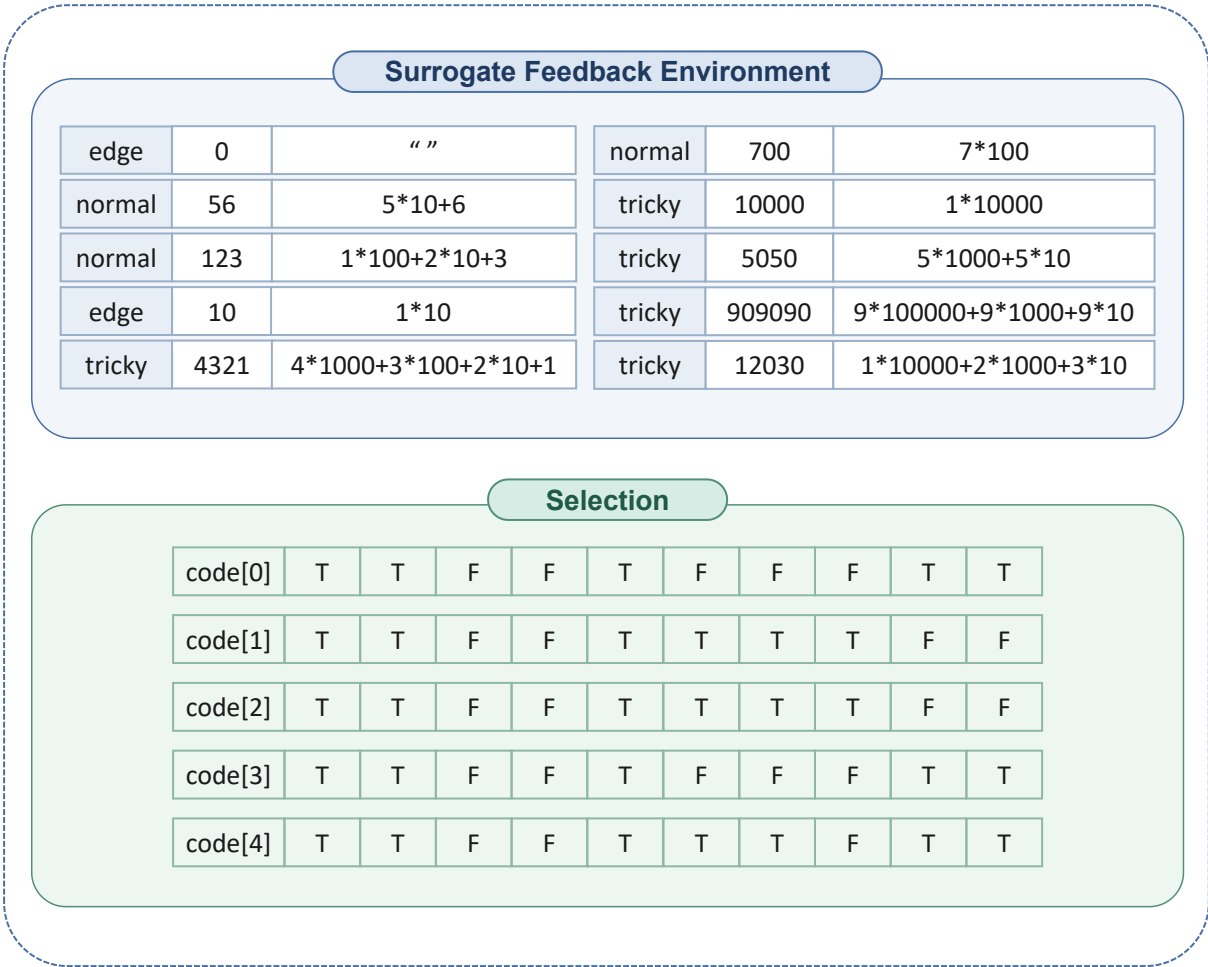


Figure 15: Surrogate tests and candidate pass/fail outcomes used for selection and contrast mining.

Semantic Spectrum Analysis

summary

SSA spectrum scan shows Bands 1–4 are semantically aligned; the first output-determining divergence appears in Band 5: the elite prints the units-place contribution with a redundant '1' (e.g., '61'), while the peer applies canonical formatting that suppresses '1' and prints the units digit as a bare literal.

fix_intent

Introduce a transferable units-place canonicalization rule without changing the elite's overall decomposition and joining strategy: when a non-zero digit contributes at exponent/power 0, emit the term as the digit alone (no explicit multiplication by 1); when power > 0, keep emitting the existing 'digit × 10^{power}' representation. This fix must operate purely at the output-representation layer (format normalization) and preserve digit extraction, zero-skipping, term ordering, and '+' concatenation.

target_location

Band 5: Termination and output formatting (units-place term canonicalization / suppress redundant '1')

must_handle

Band 1 (Input/format interpretation) comparison: both elite and peer interpret input as a single non-negative integer n and handle the degenerate $n=0$ case by returning an empty string; no Band-1 divergence explains the failure

Band 2 (Boundary conditions) comparison: both handle minimal non-zero inputs (e.g., $n=1$) and skip zero digit contributions; the failure is not an off-by-one or zero-handling error but a units-place formatting rule mismatch

Band 3 (Core state representation) comparison: both track equivalent semantic state—current digit and its place power (10^{power}); the state choice itself does not cause the discrepancy

.....

Figure 16: SSA diagnosis: earliest semantic divergence between elite and peer, summarized as a fix intent.

Lamarckian Adaptation

final code

```
def simplify(n):
    # Base case: if n is 0, return an empty string
    if n == 0:
        return ""

    # Recursive function to generate the representation
    def recursive_simplify(n, power):
        if n == 0:
            return ""
        else:
            # Extract current digit
            current_digit = n % 10
            # Recursively process the rest of the number
            rest_of_the_number = n // 10

            # Construct the current term if the digit is not zero
            current_term = ""
            if current_digit != 0:
                if power == 0:
                    current_term = f"{current_digit}"
                else:
                    current_term = f"{current_digit}{10**power}"

            # Recur for the rest of the number
            rest_terms = recursive_simplify(rest_of_the_number, power + 1)

            # Combine parts, and ensure to join with '+' properly
            if rest_terms and current_term:
                return rest_terms + "+" + current_term
            elif rest_terms:
                return rest_terms
            else:
                return current_term

    # Start the recursive function with n and power 0
    return recursive_simplify(n, 0)

# Input reading and function execution
if __name__ == "__main__":
    import sys
    input_data = sys.stdin.read().strip()
    if input_data:
        n = int(input_data)
        print(simplify(n))
```

Figure 17: Lamarckian repair: a minimal patch applied to the elite guided by the SSA fix intent.